A Unified System of Type Refinements

Jana Clara Dunfield

August 6, 2007
CMU-CS-07-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Frank Pfenning, chair
Jonathan Aldrich
Robert Harper
Benjamin Pierce, University of Pennsylvania

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

© 2007, 2022 Jana Clara Dunfield

This research was sponsored by the Air Force Research Laboratory (AFRL) under contract no. F1962895C0050, the National Science Foundation (NSF) under grant nos. CCR-0121633, CCR-0204248 and subgrant no. Y040009, and through a generous student fellowship from the NSF. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: type refinements, intersection types, union types, datasort refinements, index refinements, dimension types

Abstract

Types express properties of programs; typechecking is specification checking. However, the specifications expressed by types in conventional type systems are imprecise. Type refinements address this by allowing programmers to express more precise properties, while keeping typechecking decidable and practical.

We present a system of type refinements that unifies and extends past work on datasort and index refinements. We provide unified mechanisms of definite types, such as intersections, and indefinite types, such as unions. Through our development of *contextual typing annotations*, the *tridirectional rule*, and *let-normal typechecking*, we achieve a type system that is expressive and practical, requiring no user input besides type annotations. We show that our implementation of the type system can check several data structure invariants, as well as dimension types (an instance of *invaluable refinements*), in a subset of Standard ML.

To the ones who had a notion, a notion deep inside

Acknowledgments

I thank:

- Margaret DeLap: 11/16/01 31267;
- Frank Pfenning, one of the great all-weather advisors;
- My other committee members, particularly Benjamin Pierce, for his detailed comments;
- Håkan Younes, Mukesh Agrawal, Urs Hengartner, Aleksey Kliger, Neel R. Krishnaswami, Kate Larson, Jonathan Moody, Suman Nath, Sungwoo Park, Brigitte Pientka, Jeff Polakow, Maverick Woo, Noam Zeilberger, and other lurkers in the Wean bunker;
- Andrew Tolmach, Olivier Danvy, Rowan Davies, Tim Chevalier, and other colleagues outside the Wean bunker;
- My parents, my brother, and other relations by blood and by marriage;
- CMU Student Health Services;
- Amtrak, PAT, SEPTA, and my bicycles;
- Schenley Park in the dead of winter;
- Canada.

Contents

1 Introduction 1
1.1 Datasort refinements and intersection types . . . 2
1.2 Index refinements and union types . . . 3
1.3 The role of type annotations . . . 5
1.4 Bidirectional, tridirectional, and let-normal typechecking . . . 6
1.5 Statement of thesis . . . 7
1.6 Related work . . . 7
1.6.1 Intersection types and datasort refinements . . . 7
1.6.2 Union types . . . 10
1.6.3 Index refinements . . . 10
1.7 Other approaches . . . 12
1.7.1 Assertions . . . 12
1.7.2 ESC and related systems . . . 12
1.8 Contributions . . . 13
1.9 Reader's guide . . . 13
1.9.1 Reading online . . . 14
1.9.2 Notation . . . 14

2 A type assignment system 17
2.1 Introduction . . . 17
2.2 The base language . . . 18
2.3 Definite property types . . . 19
2.3.1 Refined datatypes . . . 20
2.3.2 Intersections . . . 21
2.3.3 Greatest type: $\top$ . . . 21
2.3.4 Index refinements and universal dependent types Π . . . 22
2.3.5 Guarded types . . . 26
2.4 Indefinite property types . . . 27
2.4.1 Unions . . . 28
2.4.2 The empty type . . . 29
2.4.3 Existential dependent types: Σ . . . 30
2.4.4 Asserting types . . . 30
2.4.5 Typechecking in evaluation order . . . 31

2.5 Properties of subtyping . . . 34
2.6 Properties of values . . . 34
2.6.1 Substitutions . . . 35
2.6.2 Definiteness . . . 36
2.6.3 Value inversion on $\rightarrow$, $*$, $\delta(i)$ . . . 41
2.6.4 Lemmas for **case** . . . 43
2.7 Type preservation and progress . . . 44
2.8 Related work . . . 48
2.9 Conclusion . . . 49

3 A tridirectional type system 51
3.1 Introduction . . . 51
3.2 The core language . . . 52
3.3 Property types . . . 56
3.3.1 Intersections . . . 56
3.3.2 Greatest type: $\top$ . . . 57
3.3.3 Refined datatypes . . . 57
3.3.4 Indefinite property types . . . 58
3.3.5 Subtyping . . . 60
3.3.6 The tridirectional rule . . . 60
3.4 Contextual typing annotations . . . 61
3.4.1 Checking against intersections . . . 61
3.4.2 Index variable scoping . . . 62
3.4.3 Contextual subtyping . . . 62
3.4.4 Soundness . . . 65
3.4.5 Completeness . . . 65
3.5 The left tridirectional system . . . 76
3.5.1 Soundness . . . 81
3.5.2 Completeness . . . 83
3.5.3 Decidability of typing . . . 87
3.5.4 Type Safety . . . 88
3.6 Related work . . . 88
3.6.1 Refinements, intersections, unions . . . 88
3.6.2 Partial inference systems . . . 88
3.6.3 Principal typings . . . 89
3.7 Conclusion . . . 89

4 Pattern matching 91
4.1 Introduction . . . 91
4.2 Foundations of pattern checking . . . 92
4.2.1 Pattern language . . . 92
4.2.2 Free variables and well-formedness . . . 92
4.2.3 Pattern matching . . . 93
4.2.4 Subtraction and intersection . . . 94

4.3 Overview 95
4.3.1 Case, match, and constructor typing 100
4.4 Type assignment version of the system 101
4.4.1 Substitution 103
4.5 Lemmas for soundness 104
4.6 Soundness 105
4.7 Limitations 110
4.8 Implementation 112
4.9 Related work 112
4.9.1 Pattern checking in unrefined type systems 112
4.9.2 Davies' datasort refinement system 112

5 A let-normal type system 115
5.1 Introduction 115
5.2 Tridirectional typechecking 116
5.2.1 Evaluation contexts do not strictly determine order 116
5.2.2 Approaching the problem 117
5.3 Let-normal typechecking 118
5.3.1 Principal synthesis of values 121
5.4 Introduction to the proofs 126
5.5 Preliminaries 127
5.6 Soundness 132
5.7 Completeness 135
5.7.1 Let-free paths and the 'precedes' relation 135
5.7.2 Properties of $\hookrightarrow$ 136
5.7.3 Position and ordering of **let**-bindings 137
5.7.4 Type preservation lemmas 138
5.7.5 Results 158
5.8 Extension to full pattern matching 166
5.9 Related work 166
5.10 Conclusion 166

6 Implementation 169
6.1 The implemented language 169
6.1.1 Type expressions 170
6.1.2 Basis 173
6.1.3 Declaring refinements 175
6.1.4 Annotating declarations 177
6.1.5 Annotating expressions 177
6.1.6 Expression syntax 178
6.2 Design 178
6.3 Initial phases 179
6.3.1 Index sort checking 179
6.3.2 Injection 179

6.3.3 Product sort flattening . . . 180
6.3.4 Subset sort elimination . . . 180
6.3.5 Let-normal translation . . . 181
6.4 Interface to an ideal constraint solver . . . 181
6.5 Constraint-based typechecking . . . 182
6.5.1 Interface to ICS . . . 184
6.5.2 Interface to CVC Lite . . . 184
6.6 Internal index domains . . . 185
6.7 Optimizations . . . 186
6.7.1 Improvement of the synthesis judgment . . . 186
6.7.2 Memoization . . . 187
6.7.3 Left rule optimizations . . . 188
6.7.4 Slack variables . . . 188
6.8 Pattern checking . . . 189
6.9 Disjunctions . . . 189
6.10 The refinement restriction . . . 191
6.11 Performance . . . 193
6.11.1 Impact of solver interfaces . . . 194
6.11.2 Conservation of speed . . . 194
6.11.3 Scaling up . . . 195
6.12 Error reporting . . . 195
6.13 Parametric polymorphism . . . 196

7 Index domains **197**
7.1 Introduction . . . 197
7.2 Integers . . . 197
7.2.1 Natural numbers . . . 198
7.2.2 Implementation . . . 198
7.2.3 Example: Inductive bitstrings . . . 198
7.2.4 Example: Red-black tree insertion . . . 204
7.2.5 Example: Red-black tree deletion . . . 209
7.3 Booleans . . . 219
7.4 Dimensions: an invaluable refinement . . . 220
7.4.1 Consistency and casting . . . 221
7.4.2 Definition of the index domain . . . 222
7.4.3 Soundness . . . 223
7.4.4 Implementation . . . 223
7.4.5 Related work on dimension types in ML . . . 224
7.4.6 Units of the same dimension . . . 226
7.4.7 Related work on invaluable refinements . . . 228
7.5 Conclusion . . . 230

8 Conclusion **233**
8.1 Future work . . . 234
8.1.1 Parametric polymorphism . . . 234
8.1.2 Refinement-based compilation . . . 238
8.1.3 Index domains . . . 239
8.1.4 Mutable references . . . 242
8.1.5 Call-by-name languages . . . 242
8.1.6 Evidence of things unseen . . . 244
8.1.7 Derivation generation . . . 245
8.1.8 Counterexample generation . . . 245
8.1.9 Suggestion tools for refinements . . . 246

A Guide to Notation **247**

Sources of Quotations **251**

Bibliography **251**

Index **265**

List of Figures

2.1 Syntax of types and terms in the initial language 19
2.2 Subtyping and typing in the initial language 19
2.3 A small-step call-by-value semantics 20
2.4 Extending the language with datatypes 20
2.5 Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$ 22
2.6 Well-formedness of types, propositions, and contexts 22
2.7 Assumed properties of the $\models$ and $\vdash$ index relations. 23
2.8 Constructor typing 25
2.9 Typing rules 32
2.10 Case typing rules 33
2.11 Subtyping rules 33
2.12 Substitution typing 35
2.13 Illustration of derivation rank 37

3.1 Syntax and semantics of the core language 55
3.2 Subtyping and typing in the core language 55
3.3 Case typing rules in the simple tridirectional system 58
3.4 Language additions for contextual typing annotations 64
3.5 Contextual subtyping 64
3.6 Examples of contextual annotations 66
3.7 Judgment forms appearing in this chapter 76
3.8 Left tridirectional system 77
3.9 Left tridirectional system, cont. 78
3.10 Case typing rules 79
3.11 Connections between our type systems 79

4.1 Definition of pattern matching 94
4.2 Operational semantics 94
4.3 Grammar 100
4.4 Typing rules for left-constructor judgments, matches, and **case** 101
4.5 Pattern typing rules 102

5.1 Syntax of terms and contexts in the let-normal type system 121

5.2 Let-normal translation 122
5.3 Rules common to the left tridirectional and let-normal type systems 128
5.4 Typing rules new in the let-normal system 129
5.5 Relation between different versions of the same direct-style term 129
5.6 Unwinding 129
5.7 Part of the new definition of the linearity judgment $\Delta \Vdash e \text{ ok}$ 130
5.8 The nearest checking position function $\langle\Downarrow\rangle(-)$ 139
5.9 Definition of the 'joins' judgment 145
5.10 The menagerie: our type systems and the key results relating them 167

6.1 Concrete syntax of the implementation language, I 170
6.2 Concrete syntax of the implementation language, II 172
6.3 Existential elimination rules for integer equations 183
6.4 Ordered binary search trees 189
6.5 Time required for typechecking 193

7.1 Examples of zippers 210
7.2 Datasort relation for zippers 211
7.3 Units of the same dimension in Stardust 227

Chapter 1

Introduction

One goal of type systems is to express program properties through types. Compile-time typechecking in statically typed languages such as ML, Haskell, and Java can catch many programmer mistakes: "well typed programs cannot 'go wrong'" [Mil78]. But there are many programs that do not behave as intended, yet do not "go wrong" in the sense of being unable to make a transition in the operational semantics. For example, in ML the type of a 'member' function on binary search trees could be written $\mathsf{tree} * \mathsf{key} \to \mathsf{bool}$. However, since this type expresses only that 'member' takes a tree and a key and a returns a Boolean, even the constant function $(\lambda t. \lambda k. \mathsf{True})$ has the above type and will pass the typechecker—to say nothing of less obviously wrong functions, such as one that descends to the left when it should descend to the right.

Hence, type safety with respect to a standard operational semantics is a necessary starting point for a static type system; it is something to know that a well-typed function will safely produce an answer of the right type (or safely diverge), but we would like to know that the answer produced is correct. That is a tall order, which we do not endeavor to satisfy—at least, not for every function in every program. We instead aim to show that the answer is not obviously wrong, and only occasionally that it is exactly right. Thus we extend conventional static typing with *property types* that encode more exact properties. Because pure type inference is, variously, undesirable or impossible in systems of this kind, the programmer must write a few more type annotations than they would in ML. We draw the line just past that additional burden: typechecking must remain automatic—ruling out interactive theorem proving—and fast enough to be done at every re-compilation.

This work unifies and extends datasort and index refinement systems, which have distinct mechanisms for combining properties: intersection types for the datasort refinements, universal and existential quantification and subset sorts for the index refinements. We provide unified mechanisms: *definite types* (including intersections and universal quantification) for the conjunction of properties, and *indefinite types* (including unions and existential quantification) for the disjunction of properties. It makes indefinite types, including union types, "first-class" by removing the need for the user to explicitly indicate when to eliminate unions. As Xi found [Xi98], the proper scope of the elimination rules for such indefinite types is not trivial; we handle this through careful formulations of both our type system and our variant of let-normal form. We show type safety in a call-by-value setting. Preliminary experiments with our implemented typechecker suggest that typechecking is practical.

In this chapter, we introduce the key concepts of datasort refinements, index refinements, and intersection and union types. Next, we state our thesis and outline the broad contours of the approach. After examining some related work (closer examinations are generally deferred to the appropriate chapters), we provide a guide to reading the rest of the dissertation.

We will view types as describing program properties that can be checked at compile time. This dissertation presents an expressive yet feasible type system with several important features: datasort refinements, intersection types, index refinements and union types, which we introduce below. We then discuss the role of type annotations in our system.

1.1 Datasort refinements and intersection types

Traditionally, datasort refinements serve to check properties that can be defined by *regular tree grammars*, a generalization of regular grammars [HU79] from strings to trees [CDG+97]. Regular tree grammars are recognized by regular tree automata; automata recognizing ordinary regular languages on strings can be seen as a special case of regular tree automata in which the trees all have the shape of a line. A very simple example is the property of a list (of integers) being of even length or odd length:

- The empty list Nil is of even length;
- If t is of even length then $\mathsf{Cons}(h, t)$ is of odd length;
- If t is of odd length then $\mathsf{Cons}(h, t)$ is of even length;
- If t is of unknown length then $\mathsf{Cons}(h, t)$ is of unknown length.

Notice that we can express the property, or *refinement*, of Cons inductively using only the property of its argument t. This corresponds to a production in a regular grammar on strings, where the right hand side has the form bB for some terminal b and nonterminal B.

Choosing to write even and odd to denote the properties of having even or odd length and list to denote the property of having unknown length, we can write the types of the constructors Nil and Cons as

$$\begin{aligned} \mathsf{Nil} &: \mathsf{even} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{even} \rightarrow \mathsf{odd} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{odd} \rightarrow \mathsf{even} \\ \mathsf{Cons} &: \mathsf{int} * \mathsf{list} \rightarrow \mathsf{list} \end{aligned}$$

We intend that Cons should have all three types listed. The type-theoretic expression of this is *intersection types*: $A \wedge B$ denotes the intersection of the types A and B, that is, the conjunction of the properties expressed by A and B. Just as $x \in S_1 \cap S_2$ implies $x \in S_1$ and $x \in S_2$, $e : A \wedge B$ implies

$e : A$ and $e : B$.

$$\begin{aligned}
\mathsf{Nil} &: \mathsf{even} \\
\mathsf{Cons} &: (\mathsf{int} * \mathsf{even} \to \mathsf{odd}) \\
&\quad \wedge (\mathsf{int} * \mathsf{odd} \to \mathsf{even}) \\
&\quad \wedge (\mathsf{int} * \mathsf{list} \to \mathsf{list})
\end{aligned}$$

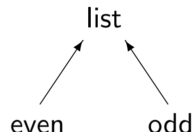
A simple function of intersection type is *tail*:

$$\begin{aligned}
\mathit{tail} \;=\; \lambda x.\, &\mathbf{case}\ x\ \mathbf{of} \\
&\quad \mathsf{Nil} \Rightarrow \mathbf{raise}\ \mathit{Error} \\
&\quad \mid \mathsf{Cons}(h, t) \Rightarrow t
\end{aligned}$$

It is easy to see that *tail* should have the type

$$\mathit{tail} \;:\; (\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{list} \to \mathsf{list})$$

Implicit in this discussion is that every value of type even (resp. odd) is known to be a value of type list. Formally, we assume the existence of a *subsort relation* $\preceq$ defining a partial order on datasorts. In this case, the subsort relation is the reflexive closure of $\{(\mathsf{even}, \mathsf{list}), (\mathsf{odd}, \mathsf{list})\}$, represented by the diagram



Prior work on datasort refinements [Fre94, Dav05a] used *sort declarations*:

$$\begin{aligned}
\mathbf{datasort}\ \mathsf{even} &= \mathsf{Nil} \mid \mathsf{Cons}\ \mathbf{of}\ \mathsf{int} * \mathsf{odd} \\
\mathbf{and}\ \mathsf{odd} &= \mathsf{Cons}\ \mathbf{of}\ \mathsf{int} * \mathsf{even}
\end{aligned}$$

which correspond to a regular grammar. In our system, the programmer gives the subsort relation and constructor types directly:

$$\mathbf{datasort}\ \mathsf{list} : \mathsf{even} < \mathsf{list}, \mathsf{odd} < \mathsf{list}$$

$$\begin{aligned}
&\mathbf{datacon}\ \mathsf{Nil} : \mathsf{even} \\
&\mathbf{datacon}\ \mathsf{Cons} : (\mathsf{int} * \mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{int} * \mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{int} * \mathsf{list} \to \mathsf{list})
\end{aligned}$$

Originally intended as an implementation shortcut, this design decision turns out to allow our system to express *invaluable* datasort refinements, which are not based on values.

1.2 Index refinements and union types

Index refinements refine types by indices drawn from a decidable constraint domain. Type systems supporting index refinements, both ours and Xi's [Xi98], are parametric in the constraint domain,

but (by far) the most explored domain is that of integers with linear inequalities, which can express a variety of useful properties, such as those involving the size of data structures. Continuing with the list example, with index refinements we can give Nil and Cons these types:

$$\begin{aligned} \mathsf{Nil} &: \mathsf{list}(0) \\ \mathsf{Cons} &: \Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{list}(a) \to \mathsf{list}(a+1) \end{aligned}$$

Here $\Pi a{:}\mathcal{N}.$ universally quantifies over an index a having *index sort* $\mathcal{N}$ (the natural numbers), so the type of Cons should be read "for any natural number a, Cons takes an integer and a list of length a and returns a list of length $a+1$". The index-refined type of the function *tail* (defined previously) is

$$\mathit{tail} \ : \ \Pi a{:}\mathcal{N}.\, (a > 0) \supset (\mathsf{list}(a) \to \mathsf{list}(a-1))$$

Here we use a *guarded type* $(a > 0) \supset \cdots$, which "guards" the rest of the type by the proposition $a > 0$. The typing claims that for all natural numbers a, if $a > 0$ holds, then *tail* takes lists of length a to lists of length $a - 1$. Thus, applications of *tail* are well typed only if $a > 0$ holds: *tail* cannot be applied to an empty list. We can now rewrite *tail*, eliminating the possibility that it raises an exception:

$$\lambda x.\ \begin{array}{l} \textbf{case}\ x\ \textbf{of} \\ \quad \mathsf{Nil} \Rightarrow \textbf{raise}\ \mathit{Error} \\ \mid \mathsf{Cons}(h,t) \Rightarrow t \end{array} \quad \longrightarrow \quad \lambda x.\ \begin{array}{l} \textbf{case}\ x\ \textbf{of} \\ \quad \mathsf{Cons}(h,t) \Rightarrow t \end{array}$$

In ordinary SML the new version of *tail* would give a "nonexhaustive match" warning, but with refinements the typechecker knows that x cannot be Nil, and concludes that the case expression is exhaustive.

To express the type of a function such as *filter*, where *filter* $f\ l$ returns the elements of l for which f returns True, we need existential quantification $\Sigma b{:}\gamma.\, B$.[1] Index refinements are not powerful enough to encode the precise length returned.

$$\mathit{filter} : \Pi a{:}\mathcal{N}.\, (\mathsf{int} {\to} \mathsf{bool}) \to \mathsf{list}(a) \to \Sigma b{:}\mathcal{N}.\, \mathsf{list}(b)$$

The binary union $A \vee B$ expresses the disjunction of the properties specified by A and B. Types of the form $\Sigma b{:}\gamma.\, B$ can be thought of as infinitary (for infinite index domains) *union types*; for example, $\Sigma b{:}\mathcal{N}.\, \mathsf{list}(b)$ corresponds informally to

$$\mathsf{list}(0) \vee \mathsf{list}(1) \vee \mathsf{list}(2) \vee \cdots$$

We call existential quantification and union types *indefinite types*. Note that while unions in function domains can be turned into intersections, e.g. $(A \vee B) \to C$ can be written $(A \to C) \wedge (B \to C)$, a union in the range of a function cannot be trivially removed. Moreover, the inclusion of union types is justified by intrinsic theoretical interest, by their symmetry with intersections, and by the similarity of the technical realizations of union types and Σ types.

[1] $\forall$ and $\exists$ instead of Π and Σ would be less obscure to many, but we follow established notation. The ASCII renderings in our implementation are `all` and `exists`.

In addition to the integer domain, we develop (in Section 7.4) an index domain of *dimensions*, allowing the type system to catch errors such as comparing or adding quantities of different dimensions (such as meters and meters squared) or different units of the same dimension (such as meters and feet). The type real of floating-point numbers is refined by the dimension index sort dim. Naturally, dimension polymorphism is supported, and both integer and dimension refinements can be used in the same program, as in the following exponentiation function

```
fun power n x =
  if  n = 0 then
    1.0
  else  if  n < 0  then
    1.0 / power (~n) x
  else
    x * power (n-1) x
```

which checks against the type $\Pi d{:}\mathsf{dim}.\, \Pi n{:}\mathcal{Z}.\, \mathsf{int}(n) \rightarrow \mathsf{real}(d) \rightarrow \mathsf{real}(d \mathbin{\hat{}} n)$.

1.3 The role of type annotations

> Type annotations are the only thing that keeps me sane.
>
> —Mukesh Agrawal

Type inference for programs without any user-provided annotations is decidable for unrefined Hindley-Milner type systems, such as core Standard ML. It is also decidable for Hindley-Milner systems extended with datasort refinements; in fact, the original work on datasorts by Freeman and Pfenning [FP91] used inference (the user must declare the arrangement of datasorts and their relationship to datatype constructors, like we did in the examples above, but need not annotate code that manipulates values of those types). For index refinements the decidability picture is less clear, though we strongly suspect that with reasonably complex domains such as the integers, inference is undecidable. But the question is moot: even if inference is possible, it is undesirable for the following reasons.

- Type annotations provide useful documentation, especially in a refined type system such as ours, since the properties are more interesting.

- Type annotations constrain components (such as functions and modules), so that these components cannot be used in ways that are *well-defined* but *unintended*. For example, one never wants to apply *tail* to an empty list, but its behavior is well-defined (it raises an exception and 'raise' expressions are always well typed) and a perfectly valid type can be inferred.

  This issue becomes especially important in large programs, and was a key factor in Davies and Pfenning's decision to *not* use inference techniques in their work on datasort refinements.[2]

[2] Inference may be desirable in cases such as nested functions, which cannot necessarily be considered "components". However, we have not tried to distinguish such cases.

- Type annotations are already required at the module level of Standard ML (where, somewhat analogously to the situation with datasort refinements, the annotations constrain how the module can be used).
- Past work on datasort refinement and index refinement systems suggests that the amount of annotation needed is modest [XP99, Dav05a], and certainly not as clumsy as in explicitly typed languages such as Java and C++.

1.4 Bidirectional, tridirectional, and let-normal typechecking

We use a form of *bidirectional typechecking* [PT98] to check annotated programs. In this scheme there are two primary "directions" of typing judgments: *checking*, $e \downarrow A$, in which it is known before e is examined that it must conform to type A, and *synthesis*, $e \uparrow A$, in which the type A is not yet known. In our formulation, the appropriate direction is closely related to the outermost syntactic form of e (for instance, abstractions $\lambda x.\, e$ never synthesize), which helps keep the typing rules simple.

Tridirectional typechecking arises in the elimination rules for unions, existentials, and the empty type. While these rules are bidirectional—their premises and conclusions consist of synthesis and checking judgments—they decompose their subject term e into $\mathcal{E}[e']$, in which e' is in an evaluation position. When such a rule is applied the typechecker's "direction" as it traverses the program is not exactly up towards the root of the syntax tree (synthesis), nor exactly down towards the leaves (propagating down a known type A to check e against, $e \downarrow A$), but moves from the subterm e' to the rest of the term based on the fact that e' is in evaluation position. We prove that if a program is well typed in our undecidable type assignment system (Chapter 2), there exists an annotated program that is type-*checkable* in the tridirectional system; conversely, if a program typechecks in the tridirectional system, we can erase the annotations and get something well typed in the type assignment system (for which we prove type safety, in Chapter 2). With intersections, unions, and index quantification, the simple form of annotation $(e : A)$ is not expressive enough, so we introduce *contextual typing annotations*. The tridirectional system is explained in Chapter 3.

Unfortunately, this "third direction" is tricky to implement—the rules of Chapter 3 allow many different decompositions of a given term. A straight implementation of the rules would, therefore, induce far too much backtracking as different decompositions are tried. Instead, our implementation performs *let-normal typechecking*. It sequentializes the program, making the order of evaluation explicit in the term syntax. The program so transformed is in *let-normal form*. Instead of rules that guess $\mathcal{E}$ such that $e = \mathcal{E}[e']$, in let-normal typechecking we replace $\mathcal{E}[e']$ with $\textbf{let}\ \overline{\mathbf{x}} = e'\ \textbf{in}\ \mathcal{E}[\overline{\mathbf{x}}]$. (Our actual transformation is more complicated than the preceding sentence might suggest.) Chapter 5 describes the transformation and the type system for terms in let-normal form, and proves that a program is well typed in the tridirectional system *if and only if* the program's let-normal version is well typed in the let-normal system. There is thus a chain of soundness and completeness results from the let-normal system to the tridirectional system (actually, between two variants of it) to the type assignment system. Therefore, the let-normal type system is an algorithmic version of the type assignment system, modulo type annotations. The curious reader may sneak a glance at the "menagerie" on page 167.

1.5 Statement of thesis

With the above background, we can state our thesis:

> *A rich type system with datasort and index refinement properties, where such properties are combined through intersection and union types, is a practical means of statically checking interesting properties of functional programs that are difficult or impossible to check in conventional static type systems.*

The dissertation supports this statement as follows. Chapter 2 presents the "rich type system", while Chapters 3, 4, and 5 go a long way toward making it "practical". Chapter 6 discusses our implementation, Stardust, furnishing evidence of practicality. Chapter 7 gives examples in two major index domains, integers and dimensions, in which interesting properties are checked.

1.6 Related work

This dissertation draws on past developments in the theory and practice of intersection types, union types, datasort refinements, and index refinements. We survey some of the work in these areas, though we postpone discussing union types until the next chapter, after we have explored the relevant technical details.

1.6.1 Intersection types and datasort refinements

Intersection types [CDCV81] characterize strong normalization [AC98][3], so type inference (for programs without type annotations) is undecidable.

Intersection types were first incorporated into a practical language by Reynolds [Rey88, Rey96], who used them to encode features such as operator overloading, and proved that typechecking programs containing intersection types is PSPACE-hard (in the worst case; we recall that ML type inference is doubly exponential in pathological cases [KMM91, KTU94], yet polynomial for the programs people actually write). Pierce [Pie91a] continued in this vein, exploring intersection types in combination with bounded polymorphism. The notion of datasort refinement combined with intersection types was introduced by Freeman and Pfenning [FP91], who showed that full type inference was decidable under the *refinement restriction* ($A \wedge B$ permitted only if A and B are refinements of the same type) and developed an inference algorithm based on techniques from abstract interpretation [CC77]. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning [DP00], who introduced a value restriction on intersection introduction, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm. Davies' datasort refinement checker, SML-CIDRE [Dav97, Dav05a, Dav05b] is built on top of the ML Kit's front end [Els05] and supports all of Standard ML.

[3] According to Kfoury [Kfo00, footnote 8], who cites a personal communication of Mariangiola Dezani, the cited work of Amadio and Curien includes the first correct published proof of this well-known result, which was apparently first correctly proved (but not published) by Venneri.

Intersection types for program analysis

Some work on program analysis in compilation uses forms of intersection and union types to infer control flow properties [WDMT02, PP01]. Because of the goals of these systems for program analysis and control flow information, the specific forms of intersection and union types are quite different from the ones considered here. As (not necessarily representative) examples, we briefly discuss System $\mathbb{I}$ and its intended successor System E.

A major goal of Kfoury and Wells' System $\mathbb{I}$ [KW04] is to allow compositional analysis through intersection types. Type inference is replaced by *typing inference*: even the context is inferred. This is intended to make the analysis truly compositional. System $\mathbb{I}$'s intersection types have a linear character: if the first occurrence of x needs to have type A and the second occurrence needs to have type B, one needs $x : A \wedge B$ in the context. Møller Neergaard and Mairson [MM04b] point out an unfortunate feature of System $\mathbb{I}$: the process of type inference corresponds exactly to normalization, so program analysis based on System $\mathbb{I}$ is precisely as useful as running the program—so why not just run the program? In contrast, our approach abandons full type inference, and appears to have a key property whose absence in System $\mathbb{I}$ is problematic: idempotency ($A \wedge A = A$, in the sense that $A \wedge A$ and A are subtypes of each other).[4] (Hence, we look askance at the authors' statement [MM04b, p. 139] that System $\mathbb{I}$ is a "representative example" of intersection type systems.)

The System E of Carlier, Wells, and others [CW04, BCKW05], intended to succeed System $\mathbb{I}$, has similar goals but a simpler technical realization. The analysis resulting from type inference in System E can correspond exactly to either call-by-value or call-by-name evaluation. A version of System E in which intersections can be variously linear or nonlinear has been formulated [CW04]. With nonlinear intersections, idempotency is gained and type inference is potentially less expensive than running the program; however, a practical type inference algorithm has not yet been presented.

Intersection types for program extraction

Through the Curry-Howard isomorphism, one can extract a program from a proof in a type theory, but the extracted programs contain logical information irrelevant to the (computational) result, making them long and inefficient. Hayashi [Hay94] developed an impredicative type theory with refinements, intersection types, and union types. His theory is designed to facilitate the exclusion of computationally irrelevant information from extracted programs.

Soft typing

Traditionally, dynamically typed languages require many so-called *runtime type checks* (more properly called tag checks). By analyzing the program, without reliance on explicit annotations, one can obtain constraints on data values that allow some of the runtime checks to be safely omitted. The basic idea is found in Reynolds [Rey67]; the name *soft typing* and a full development are due to Cartwright and Fagan [CF91]. The constraints generated involve intersection, union, and even

[4] However, it is not clear what idempotency really means for a bidirectional type system; in our system, $e \uparrow A \wedge A$ implies $e \uparrow A$ but not the converse, and there is a value restriction on intersection introduction.

conditional types [AWL94]. However, Davies [Dav05a, pp. 14–15] gives examples suggesting that datasort refinements can capture more precise invariants, due to a lack of polymorphic recursion in the soft typing system he examined (Aiken et al. [AWL94]).

To begin to compare soft typing to our approach, we should find something analogous to soft typing in a *statically* typed framework. As is well known, one can define a datatype dynamic:

$$\begin{aligned}\textbf{datatype}\ \mathsf{dynamic} = {} & \mathsf{Cons}\ \textbf{of}\ \mathsf{dynamic} * \mathsf{dynamic} \\ & \mid \mathsf{Int}\ \textbf{of}\ \mathsf{int} \\ & \mid \mathsf{Func}\ \textbf{of}\ \mathsf{dynamic} \to \mathsf{dynamic} \\ & \vdots\end{aligned}$$

Such a type can be useful in interfaces to dynamically typed software, for instance.

Soft typing, then, would seem to correspond to refinements of dynamic. Since dynamic is a perfectly ordinary datatype, it can be refined as any other in our system, and we can annotate functions on dynamic values with datasort refinements. For example, we could distinguish "dynamicized" functions from integers to integers:

$$\textbf{datasort}\ \mathsf{dynamic}\ :\ \ \mathsf{dyn_int} < \mathsf{dynamic},\ \ \mathsf{dyn_int_int_func} < \mathsf{dynamic}$$

$$\begin{aligned}&\textbf{datacon}\ \mathsf{Int} : \mathsf{int} \to \mathsf{dyn_int} \\ &\textbf{datacon}\ \mathsf{Func} : (\mathsf{dynamic} \to \mathsf{dynamic}) \to \mathsf{dynamic} \\ &\qquad\qquad \wedge\ (\mathsf{dyn_int} \to \mathsf{dyn_int}) \to \mathsf{dyn_int_int_func}\end{aligned}$$

Despite being tragically crippled by the current lack of parametric polymorphism in our system, this technique allows—and demands of—the programmer complete control over rather precise specifications, in contrast to soft typing. Note that datasorts in negative positions in constructor types, as in $((\mathsf{dyn_int} \to \mathsf{dyn_int}) \to \cdots$, present no difficulties for us: in contrast to earlier work on datasort refinements, we do not try to generate a subsort relation and constructor types from regular tree grammar-style declarations, and so avoid troublesome aspects of the semantics of subtyping [Dav05a, pp. 123–126].

Philosophical notes

> Only a moron would state $A \vee B$ if he has obtained A...
>
> —Jean-Yves Girard

This thesis largely steers clear of philosophical issues, but we briefly discuss a few here. Appropriately, we make no claim of completeness—on any level.

Product types $A * B$ can be interpreted through the Curry-Howard isomorphism as ordinary logical conjunction $A \,\&\, B$. However, the status of intersection types is less clear. We begin by noting that $\wedge$ is in a sense *more* primitive than $*$: given a few singleton types[5], say $S_1, S_2, \ldots$, we can define $A * B$ *in terms of* intersection: $A * B = (S_1 \to A) \wedge (S_2 \to B)$. (Inspired by Reynolds [Rey96], we

[5] These could be singleton types restricted to integers (as in our work), or constants analogous to the 'symbols' of Lisp and Scheme.

can also define record types in a similar style, with singletons as field labels.) However, the most general *implementation* of intersection types seems to be *as products*. For example, an intersection $(A_1 \to A_2) \wedge (B_1 \to B_2)$ can be implemented as a pair of functions (fA, fB). Thus, even if one omitted products as a primitive type constructor in (some of) a compiler's intermediate languages, they would return, possibly supplanting intersection types in the generated code.

In one of his expositions of ludics, Girard [Gir03, pp. 157–8] explains intersection types in the following way: ordinary conjunction $A \,\&\, B$ is "delocated" intersection, that is, intersection without regard for "location" (very roughly: syntax); $A \,\&\, B = \varphi(A) \cap \psi(B)$ where $\cap$ is a kind of intersection, one of Girard's primitive conjunctive operators, and φ, ψ are "delocators". Thus, intersection is a more primitive notion than conjunction. Girard points out that, computationally, intersection—unlike conjunction (computationally: product)—enjoys properties such as commutativity: $A \wedge B$ is essentially the same as $B \wedge A$. On the other hand, $A * B$ and $B * A$ are related only by canonical isomorphisms: one can readily convert values of the first to or from the second, but they are essentially different types (just as $(S_1 \to A) \wedge (S_2 \to B)$ is essentially different from $(S_1 \to B) \wedge (S_2 \to A)$, even with $\wedge$ commutative). We also observe that intersection types (as we formulate them) enjoy idempotency [MM04b], whereas A and $A * A$ are not even related by a canonical isomorphism: $(1, 1)$ might correspond to 1 but $(1, 2)$ has no corresponding value of type A.

Another subject is the semantics of refinements. Nowhere do we analyze what any particular refinement *means*. In some cases we can prove adequacy, e.g. that even corresponds to lists of even length, but the meaning of dimensions and other refinements that are not value-based is less clear. We leave all such questions to future work.

1.6.2 Union types

We discuss related work on union types in the next chapter, in Section 2.8.

1.6.3 Index refinements

DML

Xi formulated Dependent ML, a bidirectional type system [Xi98] with index refinements for a variant of ML and implemented it as an extension to Caml Light [Cam]. He showed a number of applications (using the integer constraint domain), including array bounds check elimination.

To cope with some issues around existential index quantification, Xi's approach transformed programs into a let-normal form before typechecking them; however, typechecking is then incomplete—there exist programs that typecheck in their original form but not their let-normal form. We attack similar issues with existentials in our work in a broadly similar way, through translation to our own peculiar variant of let-normal form (the subject of Chapter 5). However, our let-normal typechecking *is* complete (as well as sound, of course).

In subsequent work, Xi [Xi01, Xi02] showed that a relatively modest extension of the DML machinery suffices to statically check termination. The idea is that given a recursive function, one formulates a well-founded metric of its arguments such that the metric strictly decreases across recursive calls; it follows that those recursive calls do not cause nontermination. Reasoning involving the metric is similar to reasoning in integer-index DML.

Dependent type systems

The ancestor of index refinement is the notion of *dependent type* [AH05] developed by Martin-Löf and used in theorem proving systems such as AUTOMATH [NGd94], NuPRL [CAB+86], the Calculus of Constructions [CH88] and the Edinburgh Logical Framework [HHP93]. There, dependent types $\Pi x{:}A.\,B$ and $\Sigma x{:}A.\,B$ roughly correspond to the universal and existential quantifiers over indices; however, instead of drawing x from a restricted index domain, dependent types draw x from terms of type A. This is extremely powerful but (in any language in which some programs do not terminate) undecidable: determining if two types are equal is as hard as determining if two terms are equivalent, which cannot be done in general without evaluating them.

A number of systems have tried to tame dependent types:

- Augustsson's Cayenne language [Aug98] is an extension of Haskell with dependent types. The Cayenne typechecker "times out" if typechecking takes more than a given number of steps. The user cannot tell if the typechecker will give an answer (well-typed or not) without running it and seeing what happens.

- The Epigram system [MM04a] avoids the undecidability problem by a radical language restriction: all well-typed programs terminate, so type equivalence is decidable. The dependent indices are elements of *inductive families* of constructors; the example of natural numbers with *zero* and *succ* constructors is probably the canonical one.

- In the dependent type system of Chen and Xi [CX05], as in Epigram, users can write explicit proofs of type equivalences, with inductive families of constructors as the means of constructing indices. However, unlike Epigram, the language itself is not restricted—decidability comes by restricting the terms that can inhabit indices.

- Licata and Harper [LH05] present a system that, like Chen and Xi's, allows explicit proofs of type equivalence with inductive families. In contrast to Chen and Xi, their system supports functions on indices as well as index propositions; instead of a relation $mul(i, j, k)$ that holds if i times j equals k, one can simply write $mul(i, j)$ in Licata and Harper's system. These functions on indices can operate not only during typechecking but at runtime, to "retype" terms with new indices, in contrast to Chen and Xi where indices are exclusively a compile-time construct.

Other similar type systems are found in the evolving "Ωmega" language of Sheard et al. [She04, SP04] and Westbrook et al.'s RSP1 [WSW05]. For a detailed comparison of these systems, see Licata and Harper [LH05]; for an exposition of dependent types with historical notes, see Aspinall and Hofmann [AH05].

We believe our approach has two major advantages over these systems. The first is that our system needs no guidance beyond type annotations. The second is the legibility and clarity of the types themselves. We believe that all the types in our system are easy to understand—significantly more so than in the dependent type systems. (We do not claim, at least not here, that our type system *per se* is easy to understand, merely that the types written and read by users are.) This is a subjective issue, but these systems strike us as too clever by half. It could be argued that both flavors of system add to the number of 'levels' a user must think about—ours adds index refinements (and

datasorts, but let us not muddy the comparison), while theirs add dependent typing and kind-level programming.[6] However, the level we add seems to be conceptually *inferior* to the types in conventional type systems, rather than equal or superior, and this may play a role in the usability gap we allege.

1.7 Other approaches

In this section, we briefly examine a few other approaches to the problem of verifying software properties, and compare them to type refinements.

1.7.1 Assertions

Assertion mechanisms allow any Boolean expression to be tested at runtime. Such assertions serve as documentation and also encourage "early failure". Of course, the lack of restrictions on the form of the asserted expression makes compile-time assertion checking undecidable.

We can view a function type $A \to B$ as an implicit pair of assertions: a precondition of the form "the argument is of type A" and a postcondition of the form "the result is of type B". With type refinements, pre- and postconditions of this form are more expressive than in a conventional static type system. Thus, type refinements reduce but do not eliminate the utility of generic assertion facilities such as Findler and Felleisen's [FF02].

Note that under certain circumstances, a compiler might optimize away the assertion check: if dataflow analysis shows that $x = 1$ when $\mathit{assert}(x > 0)$ is reached, the assertion is equivalent to $\mathit{assert}(\mathtt{true})$ which is a no-op. However, most compilers provide little feedback as to which checks are actually removed, so this is still a very poor substitute for static checking.

1.7.2 ESC and related systems

ESC, the Extended Static Checker [DLNS98, Lei01] is intended to help find bugs in Modula-3 and Java programs. Annotations can be written expressing bounds on integers and synchronization properties, among others. ESC then attempts to check the program against the annotations. ESC is *unsound*: it may incorrectly report that there are no problems. In practice, though, the system has been good at finding bugs.

Type refinements are less expressive than the property language of ESC. However, even if we restrict our attention to a subset of the ESC properties that can be expressed through type refinements—for example, a pointer being non-null corresponds to a value of `option` type having a particular datasort refinement—ESC can get away with having less information (annotations), since it uses several tricks to (unsoundly) finesse undecidable problems such as generating loop invariants. The designers of ESC argue that much can be gained by giving up the "shackles of soundness" [DLNS98, p. 33]; we believe that the present work, and the past work on type refinements, have only begun to show how far one can go *without* giving up soundness. It is interesting

[6] If we had to characterize the 'level' of intersections and unions, we would place them alongside types, or if a refinement restriction is enforced, at an inferior level (a distinction consistent with Davies' "intersection sorts").

to note that Leino [Lei01], in his discussion of "future challenges", suggests exploring "more-than-types systems"—by which he means roughly what we mean by type refinements, though he focuses on imperative languages rather than functional languages.

In a similar vein, the Spec# system [BLS04] can check object invariants in C# programs; unlike ESC, it is based on a sound methodology [BDF+04].

1.8 Contributions

The major contributions of this dissertation are:

- An elegant formulation of union types and existential index quantification, without clumsy syntactic markers for elimination.
- A systematic formulation of bidirectional typechecking, with a new kind of type annotation suited to our rich type language.
- A new variant of the let-normal transform, by which we make it practical to check unions and existentials.
- A formulation of dimension types as an index domain.
- A typechecker for a subset of SML, including all the features above.

1.9 Reader's guide

This chapter ends with a description of some of the notation used subsequently (Section 1.9.2). The rest of the thesis is organized as follows:

Chapter 2	describes a type assignment system for property types, including datasort and index refinements, intersections, and unions;
Chapter 3	describes a decidable "tridirectional" formulation of the type assignment system;
Chapter 4	describes a type system for a richer language of patterns than that considered in Chapters 2, 3 and 5;
Chapter 5	describes a more tractable "let-normal" formulation of the tridirectional system;
Chapter 6	describes an implementation of a version of the let-normal system (including the richer patterns of Chapter 4);
Chapter 7	discusses the constraint domains for index refinements that we have implemented, including dimensions as an illustrative example of *invaluable refinements* (refinements not based on values);

Chapter 8	concludes, considering future work;
Appendix A	provides a glossary of notation.

Completists can simply read the chapters in sequence. Those interested only in the theory of intersection and union types may read just Chapter 2. Implementors might skim Chapter 2 but read the rest of the work closely, except for the proofs. Finally, functional programmers with less interest in theory should focus on Chapters 6 and 7.

The chapters on pattern matching and index domains (4 and 7, respectively) can be skipped without substantially impeding understanding of other chapters; however, without the examples in Chapter 7 some of our work's motivation will be missed.

The reader should have a solid grasp of type theory as applied to programming languages [Pie02]; some familiarity with natural deduction and the sequent calculus [Pra65] is helpful but not mandatory.

1.9.1 Reading online

The PDF of this dissertation includes the usual hyperlinks (section references, citations, and URLs) as well as reverse links from bibliography entries to the citing section. Moreover, the names of implementation examples are links to files on www.type-refinements.info (which the author expects to maintain indefinitely).

1.9.2 Notation

We mention some of our notational conventions here; a more complete guide can be found in Appendix A.

Capital letters A, B, C, D stand for types; P stands for propositions. e is used for expressions (or terms; we use the two words interchangeably). *Script letters* stand for terms with holes: for example, $\mathcal{E}$ is used for evaluation contexts and $\mathcal{C}$ for a term containing a hole in any position. Two exceptions are script $\mathcal{D}$, which stands for derivations, and script $\mathcal{R}$, which stands for the name of a rule of inference; we write $\mathcal{D} :: \Gamma \vdash e : A$ for "$\mathcal{D}$ derives the judgment $\Gamma \vdash e : A$", and can speak of "the rule $\mathcal{R}$ that concludes $\mathcal{D}$".

Bold letters $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$, etc. are used for linear variables.

Lowercase letters a, b (and others) stand for index variables; c stands for constructors of inductive datatypes; i, j, k stand for index expressions (but also for mathematical indices: "for all k in $1..n$").

Proof notation

Many of our proofs are presented in line-by-line format (without numbering). The proposition or judgment appears on the left with its justification (if not obvious) on the right.

- In proofs by induction, "IH" refers to the induction hypothesis. If the result to be proved is in several parts, numbered say (1), (2), (3), then "IH (2)" refers to the second part (not to all parts of the induction hypothesis, applied twice, which would be "IH (twice)").

- Most steps use results from one of the preceding few lines, or in any line concluding a "block". A schematic example:

A1	Given
A2	Given
A1 and A2	
B1	
B2	By Thm. Such-and-such *(which says that B1 implies B2)*

  (A1 and A2) and B2

  Here, "A1 and A2" and "B2" conclude nontrivial blocks, while "(A1 and A2) and B2" concludes a block containing only itself.

  This convention is not followed strictly, but when looking for the antecedents to a result one should look first directly above the result, then at lines concluding blocks.

- In proofs where the desired result has several parts (such as the proof of Lemma 5.60), each part of the result, when obtained, is marked "☞". This is useful since in many cases, part of the result is obtained soon (e.g. upon application of the induction hypothesis) while the rest of the result requires several more steps; this convention allows us to avoid restating the parts previously shown.

  When reasoning equationally over several lines, "☞" may appear on the last line with the left side of the equation elided. For example, if part of the result to be proved is that $\mathcal{E} = \mathcal{E}'$, we might write this proof, in which "☞" highlights that part of the result.

	$\mathcal{E}_0 = [\,]$	Given
	$\mathcal{E} = \mathcal{E}'[\mathcal{E}_0]$	Given
	$= \mathcal{E}'[[\,]]$	By $\mathcal{E}_0 = [\,]$
☞	$= \mathcal{E}'$	By defn. of evaluation contexts
	$\vdots$	
☞	*the rest of the result to be proved*	

Chapter 2

A type assignment system[1]

In this chapter, we develop a system of type assignment with intersection types, union types, indexed types, and universal and existential dependent types that is sound in a call-by-value functional language. The combination of logical and computational principles underlying our formulation naturally leads to the central idea of typechecking subterms in evaluation order. We thereby provide a uniform generalization and explanation of several earlier isolated systems. While undecidable, this system is the basis for the decidable system in Chapter 3, the practical system in Chapter 5, and ultimately the typechecker described in Chapter 6.

The system in this chapter is a Curry-style type assignment system. Our goal is to typecheck code, not to compile it. Elaborating the source language into a form suitable for use as a typed intermediate language—one with explicit polymorphic instantiation, for example—would have few, if any, advantages in this typechecking-focused setting. In this respect, we follow Davies' datasort refinement system [Dav05a], which is also focused on typechecking rather than compilation. Thus, we see no compelling arguments against type assignment in this setting, and prior work suggests that it is a solid foundation for type refinement systems.

2.1 Introduction

Conventional static type systems are tied directly to the expression constructs available in a language. For example, functions are classified by function types $A \rightarrow B$, pairs are classified by product types $A * B$, and so forth. In more advanced type systems we find type constructs that are independent of any particular expression construct. The best-known examples are parametric polymorphism $\forall\alpha. A$ and intersection polymorphism $A \wedge B$. Such types can be seen as expressing more complex properties of programs. For example, if we read the judgment $e : A$ as e *satisfies property* A, then $e : A \wedge B$ expresses that e satisfies both property A and property B. We call such types *property types*. The aim is to integrate a rich system of property types into practical languages such as Standard ML [MTHM97], in order to express and verify detailed invariants of programs as part of typechecking.

In this chapter we design a system of property types specifically for call-by-value languages.

[1]This chapter is based on joint work with Frank Pfenning [DP03].

We show that the resulting system is type-safe, that is, satisfies the type preservation and progress theorems. We include indexed types $\delta(i)$, intersection types $A \wedge B$, a greatest type $\top$, universal dependent types $\Pi a{:}\gamma.\, A$, union types $A \vee B$, an empty type $\bot$, and existential dependent types $\Sigma a{:}\gamma.\, A$. We thereby combine, unify, and extend prior work on intersection types [DP00], union types [Pie91b, BDCd95] and index refinements [Xi98, XP99].

Several ideas emerge from our investigation. Perhaps most important is that type assignment may visit subterms in evaluation order, rather than just relying on immediate subterms. We also confirm the critical importance of a logically motivated design for subtyping and type assignment. The resulting orthogonality of various property type constructs greatly simplifies the theory and allows one to understand each concept in isolation. As a consequence, simple types, intersection types [DP00], and indexed types [XP99] are extended *conservatively*. The type system is designed to allow effects (in particular, mutable references), but in order to concentrate on more basic issues, we do not include them explicitly (see [DP00] for the applicable techniques to handle mutable references).

The system of pure type assignment presented in this chapter is undecidable. Chapter 3 presents a decidable version based on bidirectional typechecking (in the style of [DP00, Dun02]) of programs containing some type annotations, where we variously *check* an expression against a type or else *synthesize* the expression's type.

The remainder of the chapter is organized as follows. We start by defining a small and conventional functional language with subtyping, in a standard call-by-value semantics. We then add several forms of property types: intersection types, indexed types, and universal dependent types. As we do so, we motivate our typing and subtyping rules through examples, showing how our particular formulation arises out of our demand that the theorems of *type preservation* and *progress* hold. Then we add the *indefinite* property types: the empty type $\bot$, the union type $\vee$, and the existential dependent type Σ. To be sound, these must visit subterms in evaluation order. After proving some novel properties of judgments and substitutions, we prove preservation and progress. Finally, we discuss related work and conclude.

2.2 The base language

We start by defining a standard call-by-value functional language (Figure 2.1) with functions, a unit type (used in a few examples), products, and recursion, to which we will add various constructs and types. Expressions do not contain types, because we are formulating a pure type assignment system. We distinguish between variables x that stand for values and variables u that stand for expressions, where the latter arise only from fixed points $\textbf{fix}\ u.\, e$. The form of the typing judgment is

$$\Gamma \vdash e : A$$

where Γ is a context typing variables x and u. The typing rules so far are standard (Figure 2.2); the subsumption rule utilizes a subtyping judgment

$$\Gamma \vdash A \leq B$$

meaning that A is a subtype of B in context Γ. The interpretation is that the set of values of type A is a subset of the set of values of type B. The context Γ is not used in the subtyping rules of Figure

$$A, B, C, D ::= \mathbf{1} \mid A \to B \mid A * B$$
$$e ::= x \mid u \mid () \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{fix}\, u.\, e \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e)$$

Figure 2.1: Syntax of types and terms in the initial language

$$\frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \to \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ \mathbf{1} \qquad \frac{\Gamma \vdash A_1 \leq B_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 * A_2 \leq B_1 * B_2} *$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ \mathsf{var} \qquad \frac{\Gamma(u) = A}{\Gamma \vdash u : A}\ \mathsf{fixvar} \qquad \frac{\Gamma, u{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\, u.\, e : A}\ \mathsf{fix}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\ \mathsf{sub}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B} \to\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B} \to\mathrm{E}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\ \mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 * A_2} *\mathrm{I} \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{fst}(e) : A} *\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{snd}(e) : B} *\mathrm{E}_2$$

Figure 2.2: Subtyping and typing in the initial language

2.2, but we subsequently augment the subtyping system with rules that refer to Γ. The rule $\to$ is the standard subtyping rule for function types, contravariant in the argument and covariant in the result; rule $\mathbf{1}$ is obvious. It is easy to prove that subtyping is decidable, reflexive ($\Gamma \vdash A \leq A$), and transitive (if $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$); as we add rules to the subtyping system, we maintain these properties.

The subtyping rules for our system are designed following the well-known principle that $A \leq B$ only if any (closed) value of type A also has type B. Thus, whenever we must check if an expression e has type B we are safe if we can synthesize a type A and $A \leq B$. The subtyping rules then naturally decompose the structure of A and B by so-called *left* and *right* rules that closely mirror the rules of a sequent calculus [Pra65, Appendix A].

A call-by-value operational semantics defining a relation $e \mapsto e'$ is given in Figure 2.3. We use v for values, and write $e\ \mathsf{value}$ to mean that e is a value. We write $\mathcal{E}$ for an evaluation context—a term containing a hole $[]$; we write $\mathcal{E}[e']$ to denote $\mathcal{E}$ with its hole replaced by e'.

2.3 Definite property types

Definite types accumulate positive information about expressions. For instance, the intersection type $A \wedge B$ expresses the conjunction of the properties A and B. We later introduce *indefinite types* such as $A \vee B$ which encompass expressions that have either property A or property B, although it is unknown which one.

$$\begin{array}{rl} \text{Values} & v ::= x \mid () \mid \lambda x.\, e \mid (v_1, v_2) \\ \text{Evaluation contexts} & \mathcal{E} ::= [\,] \mid \mathcal{E}\, e \mid v\, \mathcal{E} \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) \mid \mathbf{fst}(\mathcal{E}) \mid \mathbf{snd}(\mathcal{E}) \end{array}$$

$$\frac{e' \mapsto_R e''}{\mathcal{E}[e'] \mapsto \mathcal{E}[e'']}\ \text{ev-context} \qquad \begin{array}{rcl} (\lambda x.\, e)\, v & \mapsto_R & [v/x]\, e \\ \mathbf{fix}\, u.\, e & \mapsto_R & [(\mathbf{fix}\, u.\, e) \,/\, u]\, e \\ \mathbf{fst}(v_1, v_2) & \mapsto_R & v_1 \\ \mathbf{snd}(v_1, v_2) & \mapsto_R & v_2 \end{array}$$

Figure 2.3: A small-step call-by-value semantics

$$\begin{array}{rl} ms ::= & \cdot \mid\ c(x) \Rightarrow e \mid ms \\ e ::= & \ldots \mid c(e) \mid \mathbf{case}\ e\ \mathbf{of}\ ms \\ v ::= & \ldots \mid c(v) \\ \mathcal{E} ::= & \ldots \mid c(\mathcal{E}) \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ ms \end{array}$$

$$\mathbf{case}\ c(v)\ \mathbf{of}\ \ldots c(x) \Rightarrow e \ldots \mapsto_R [v/x]\, e$$

Figure 2.4: Extending the language with datatypes

2.3.1 Refined datatypes

We now add datatypes with refinements (Figure 2.4). $c(e)$ denotes a datatype constructor c applied to an argument e; the destructor **case** e **of** ms denotes analysis of e with one layer of non-redundant and exhaustive matches ms, each of the form $c_k(x_k) \Rightarrow e_k$. For example, the only permitted **case** expression on the list type is **case** e **of** $\mathsf{Nil}(x_1) \Rightarrow e_1 \mid \mathsf{Cons}(x_2) \Rightarrow e_2$, in which $x_1 : \mathbf{1}$ and $x_2 : \mathsf{int} * \mathsf{list}$. This restricted language of patterns will be enlarged in Chapter 4.

Each datatype is refined by an *atomic subtyping* relation $\preceq$ over *datasorts* δ. Each datasort identifies a subset of values of the form $c(v)$, yielding definite information about a value. For example, datasorts true and false identify singleton subsets of values of the type bool.

A new subtyping rule defines subtyping for datasorts in terms of the atomic subtyping relation $\preceq$:

$$\frac{\delta_1 \preceq \delta_2}{\Gamma \vdash \delta_1 \leq \delta_2}\ \delta$$

To maintain reflexivity and transitivity of subtyping, we require the same properties of atomic subtyping: $\preceq$ must be reflexive and transitive.

Since we will subsequently further refine our datatypes by indices, we defer discussion of the typing rules.

2.3.2 Intersections

The typing $e : A \wedge B$ expresses that e has type A and type B. The subtyping rules for $\wedge$ capture this:

$$\frac{\Gamma \vdash A \leq B_1 \quad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2} \wedge R \quad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B} \wedge L_1 \quad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B} \wedge L_2$$

We omit the common distributivity rule

$$\frac{}{(A \to B) \wedge (A \to B') \ \leq \ A \to (B \wedge B')}$$

which Davies and Pfenning showed to be unsound in the presence of mutable references [DP00]. The present system does not have mutable references, but is intended to be compatible with such, as well as with other forms of effectful computation; Davies [Dav05a, pp. 54–55] analyzes distributivity in the more general context of effects captured by the $\bigcirc$ modality [Mog88]. Another reason to omit the distributivity rule is that without it, no subtyping rule contains more than one type constructor: the rules are orthogonal. As we add type constructors and subtyping rules, we maintain this orthogonality. In fact, the left and right subtyping rules for $\wedge$ and the other property types closely mirror the left and right rules of a sequent calculus [Pra65, Appendix A]. Ignoring Γ, we can think of subtyping as a single-antecedent, single-succedent form of the sequent calculus.

On the level of typing, we can introduce an intersection with the rule

$$\frac{\Gamma \vdash v : A_1 \quad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2} \wedge I$$

and eliminate it with

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1} \wedge E_1 \quad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2} \wedge E_2$$

Note that $\wedge I$ can only type values v, not arbitrary expressions, following Davies and Pfenning [DP00] who showed that in the presence of mutable references, allowing non-values destroys type preservation.

The $\wedge$-elimination rules are derivable via rule sub with the $\wedge L_{1,2}$ subtyping rules. However, we include them because they are not derivable in a bidirectional system such as that of Chapter 3.

2.3.3 Greatest type: $\top$

It is easy to incorporate a greatest type $\top$, which can be thought of as the 0-ary form of $\wedge$. The rules are simply

$$\frac{}{\Gamma \vdash A \leq \top} \top R \qquad \frac{\Gamma \vdash v \ \mathsf{ok}}{\Gamma \vdash v : \top} \top I$$

There is no left subtyping rule. The typing rule is essentially the 0-ary version of $\wedge I$, the rule for binary intersection. The premise $\Gamma \vdash v\ \mathsf{ok}$ says that the free variables of e are in $\mathrm{dom}(\Gamma)$. Without this premise, we could derive $\cdot \vdash y : \top$, where y is an unknown identifier. Such anomalies would be problematic in Chapter 5. Finally, note that if we allowed $\top I$ to type non-values, the progress theorem would fail: $\vdash ()\,() : \top$, but $()\,()$ is neither a value nor a redex.

$$P ::= \bot \mid i \doteq j \mid \ldots$$
$$\Gamma ::= \cdot \mid \Gamma, x{:}A \mid \Gamma, u{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P$$

$$\begin{aligned} \overline{\cdot} &= \cdot \\ \overline{\Gamma, x{:}A} &= \overline{\Gamma} \\ \overline{\Gamma, a{:}\gamma} &= \overline{\Gamma}, a{:}\gamma \\ \overline{\Gamma, P} &= \overline{\Gamma}, P \end{aligned}$$

Figure 2.5: Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

$\boxed{\Gamma \vdash A \text{ wf}}$ $\qquad$ $\boxed{\Gamma \vdash P \text{ wf}}$

$$\frac{FV(A) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash A \text{ wf}} \qquad \frac{FV(P) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash P \text{ wf}}$$

$\boxed{\Gamma \text{ wf}}$

$$\frac{}{\cdot \text{ wf}} \qquad \frac{\Gamma \text{ wf} \qquad x \notin \text{dom}(\Gamma) \qquad \Gamma \vdash A \text{ wf}}{\Gamma, x{:}A \text{ wf}}$$

$$\frac{\Gamma \text{ wf} \qquad a \notin \text{dom}(\Gamma)}{\Gamma, a{:}\gamma \text{ wf}} \qquad \frac{\Gamma \text{ wf} \qquad \Gamma \vdash P \text{ wf}}{\Gamma, P \text{ wf}}$$

Figure 2.6: Well-formedness of types, propositions, and contexts

2.3.4 Index refinements and universal dependent types Π

Now we add index refinements, which are dependent types over a restricted domain, closely following Xi and Pfenning [XP99], Xi [Xi98, Xi00], and Dunfield [Dun02]. This refines datatypes not only by datasorts, but by indices drawn from some constraint domain: the type $\delta(i)$ is the refinement by δ and index i.

To accommodate index refinements, several changes must be made to the systems we have constructed so far. The most drastic is that Γ can include *index variables* a, b and propositions P as well as program variables. Because the program variables are irrelevant to the index domain, we can define a *restriction function* $\overline{\Gamma}$ that yields its argument Γ without program variable typings (Figure 2.5). No variable may appear twice in Γ, but ordering of the variables is now significant because of dependencies. That is, in the context $\Gamma_1, x{:}\mathsf{list}(a), \Gamma_2$, the index variable a must be declared $(a{:}\gamma)$ in Γ_1—contexts such as $x{:}\mathsf{list}(a), a{:}\mathcal{N}$ are ill-formed. These requirements are made explicit in rules for the *well-formedness* of types, propositions, and contexts (Figure 2.6). However, we do not refer to these rules except implicitly, as we assume throughout the thesis that all types, propositions, and contexts are well formed. For instance, given the context $\Gamma_1, a{:}\gamma, \Gamma_2$, we know from the implicit $(\Gamma_1, a{:}\gamma, \Gamma_2)$ wf that $a \notin \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)$ and $a \notin FV(\Gamma_1)$.

Our formulation, like Xi's, requires only a few properties of the constraint domain: There must be a way to decide a consequence relation

$$\overline{\Gamma} \models P$$

whose interpretation is that given the index variable typings and propositions in $\overline{\Gamma}$, the proposition

Property 2.2 (Substitution). *Given* $\overline{\Gamma_1} \vdash i : \gamma$*:*

(i) *If* $\overline{\Gamma_1}, a{:}\gamma, \overline{\Gamma_2} \models P$ *then* $\overline{\Gamma_1}, [i/a]\overline{\Gamma_2} \models [i/a]\,P$.

(ii) *If* $\overline{\Gamma_1}, a{:}\gamma, \overline{\Gamma_2} \vdash j : \gamma'$ *then* $\overline{\Gamma_1}, [i/a]\overline{\Gamma_2} \vdash [i/a]\,j : \gamma'$.

Property 2.3 (Weakening). *Given* $(\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2})$ wf*:*

(i) *If* $\overline{\Gamma_1}, \overline{\Gamma_2} \vdash i : \gamma$ *then* $\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2} \vdash i : \gamma$.

(ii) *If* $\overline{\Gamma_1}, \overline{\Gamma_2} \models P$ *then* $\overline{\Gamma_1}, \overline{\Gamma}, \overline{\Gamma_2} \models P$.

Property 2.4 (Equivalence). *If* $\overline{\Gamma} \vdash i : \gamma$ *and* $\overline{\Gamma} \vdash j : \gamma$ *and* $\overline{\Gamma} \vdash k : \gamma$ *then*

(i) *The relation* $\overline{\Gamma} \models i \doteq i$ *holds.*

(ii) *If* $\overline{\Gamma} \models i \doteq j$ *then* $\overline{\Gamma} \models j \doteq i$.

(iii) *If* $\overline{\Gamma} \models i \doteq j$ *and* $\overline{\Gamma} \models j \doteq k$*, then* $\overline{\Gamma} \models i \doteq k$.

Property 2.5. *The relation* $\cdot \models \bot$ *does not hold.*

Property 2.6. $\overline{\Gamma}, a{:}\gamma \vdash a : \gamma$.

Property 2.7 (Consequence). *If* $\overline{\Gamma} \models P_k$ *for all* $P_k \in \{P_1, \ldots, P_n\}$*, and* $\overline{\Gamma}, P_1, \ldots, P_n \models P'$ *then* $\overline{\Gamma} \models P'$.

Figure 2.7: Assumed properties of the $\models$ and $\vdash$ index relations.

P must hold. Among the propositions must be $i \doteq j$, denoting equality. There must be a way to decide a relation

$$\overline{\Gamma} \vdash i : \gamma$$

whose interpretation is that i has *sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have sorts; terms and indices are distinct. Our proofs require that $\models$ be a consequence relation (that is, if some assumption in $\overline{\Gamma}$ is entailed by the rest of $\overline{\Gamma}$, it can be removed without changing what is entailed), that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that both $\models$ and $\vdash$ have obvious substitution and weakening properties (Figure 2.7).

Remark 2.1. The system in this chapter is undecidable, so there is no fundamental reason any of these relations must be decidable; the requirement becomes necessary in Chapter 3.

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. In our examples, we work in a domain of integers $\mathcal{N}$ with $\doteq$ and some standard operations ($+$, $-$, $*$, $<$, and so on); each datatype is refined by indices of sort $\mathcal{N}$. Then $\overline{\Gamma} \models P$ is decidable provided the inequalities in P are linear.

We add an infinitary definite type $\Pi a{:}\gamma.\, A$, introducing an index variable a universally quantified over indices of sort γ. One can also view Π as a dependent product restricted to indices (instead of arbitrary terms). We also add a *guarded type* $P \supset A$, read "P implies A", discussed below.

Example. Assume we define a datatype of integer lists: a list is either $\mathsf{Nil}()$ or $\mathsf{Cons}(h, t)$ for some integer h and list t. Refine this type by a datasort odd of odd-length lists, and by a datasort even of even-length lists. We also refine the lists by their length, so Nil has type $\mathbf{1} \to \mathsf{even}(0)$, and Cons has type $(\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{even}(a) \to \mathsf{odd}(a+1)) \wedge (\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{odd}(a) \to \mathsf{even}(a+1))$. Then the function

$$\textbf{fix}\ \textit{repeat}.\ \lambda x.\ \textbf{case}\ x\ \textbf{of}\ \mathsf{Nil}() \Rightarrow \mathsf{Nil}()\ \mid\ \mathsf{Cons}(h, t) \Rightarrow \mathsf{Cons}(h, \mathsf{Cons}(h, \textit{repeat}(t)))$$

will have type $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$.

To handle the indices, we modify the subtyping rule δ from Section 2.3.1 so that it checks (separately) the datasorts δ_1, δ_2 and the indices i, j:

$$\frac{\delta_1 \preceq \delta_2 \qquad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ \delta$$

Datatype constructors c are typed by a judgment

$$\Gamma \vdash c : A^{con}$$

where

$$\text{Constructor types}\quad A^{con}\ ::=\ B \to \delta(i) \mid A^{con} \wedge A^{con} \mid \Pi a{:}\gamma.\, A^{con} \mid P \supset A^{con}$$

That is, the type of a constructor is $B \to \delta(i)$ where B is unrestricted, or an intersection, universal quantification, or guard of such a type. The only rule deriving $\Gamma \vdash c : A^{con}$ is $\mathcal{S}$-con, which uses a *synthesis subtyping* judgment $\Gamma \vdash A \uparrow B$ (see Figure 2.8). Synthesis subtyping is essentially a weaker form of subtyping with A as input and B as output: if $A \uparrow B$ then $A \leq B$, but not the converse. For example, $A \leq \top$ but $A \not\uparrow \top$.

We assume a *constructor signature* $\mathcal{S}$ that gives types to constructors. In the implementation, this is given explicitly by the user; see Chapter 6.[2]

The rule for constructor application is

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \qquad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\ \delta\mathrm{I}$$

To type **case** e **of** ms where $e : \delta(i)$, rule δE checks that all the case arms in ms have the same type C, expressed by the premise $\Gamma \vdash ms :_{\delta(i)} C$, read "$ms$ checks against C, assuming the expression cased upon has type $\delta(i)$".

$$\frac{\Gamma \vdash e : \delta(i) \qquad \Gamma \vdash ms :_{\delta(i)} C}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms : C}\ \delta\mathrm{E}$$

To check an arm $c(x) \Rightarrow e$, we analyze the type $\mathcal{S}(c)$ of the constructor, accounting for all the ways c could have been applied to create a value of type $\delta(i)$. For example, if $\mathcal{S}(c) = (A_1 \to \delta_1(i_1)) \wedge (A_2 \to \delta(i_2))$, we need to check $e : C$ with two obligations: in the first we assume $x{:}A_1, i_1 \doteq i$, and in the second $x{:}A_2, i_2 \doteq i$. We introduce a judgment form

$$\Gamma; c : A^{con}; c(x) : B \vdash e : C$$

[2]This approach differs materially from Davies' system [Dav05a], which *generates* constructor types from a regular tree grammar; see Section 7.4.7.

$\boxed{\Gamma \vdash A \uparrow B}$

$$\frac{}{\Gamma \vdash A \uparrow A}\ \text{refl-}\uparrow \qquad \frac{\Gamma \vdash A \uparrow A'}{\Gamma \vdash A \wedge B \uparrow A'}\ \wedge\uparrow_1 \qquad \frac{\Gamma \vdash B \uparrow B'}{\Gamma \vdash A \wedge B \uparrow B'}\ \wedge\uparrow_2$$

$$\frac{\Gamma \vdash [i/a]A \uparrow A' \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \uparrow A'}\ \Pi\uparrow \qquad \frac{\Gamma \vdash A \uparrow A' \qquad \overline{\Gamma} \models P}{\Gamma \vdash P \supset A \uparrow A'}\ \supset\uparrow$$

$\boxed{\Gamma \vdash c : A^{con}}$

$$\frac{\Gamma \vdash \mathcal{S}(c) \uparrow A}{\Gamma \vdash c : A}\ \mathcal{S}\text{-con}$$

Figure 2.8: Constructor typing

which is read "under Γ, assuming c has the constructor type A^{con} and $c(x)$ (which is the value cased upon) has type B, the case arm e has type C".

$$\frac{}{\Gamma \vdash \cdot :_B C}\ \mathsf{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e : C \qquad \Gamma \vdash ms :_B C}{\Gamma \vdash (c(x) \Rightarrow e \mid ms) :_B C}\ \mathsf{casearm}$$

Rules δS-ct and δF-ct cover the "base case" in which the type of c is simply $A \to \delta(i)$. In each rule, we have assumptions that a constructor c has type $A \to \delta(i)$ and a value $c(x)$ has type $\delta'(i')$.

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i' \vdash e : C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta' \qquad \Gamma, x{:}A \vdash e\ \mathsf{ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{F-ct}$$

If $\delta \not\preceq \delta'$ (rule δF-ct), those assumptions are inconsistent: either

- δ and δ′ are incomparable (neither $\delta \preceq \delta'$ nor $\delta' \preceq \delta$), so clearly $c : A \to \delta(i)$ cannot produce a $\delta'(i')$ when applied to x, or

- $\delta' \preceq \delta$ (but $\delta \not\preceq \delta'$); here, observe that $c : A \to \delta(i)$ represents *all* the information we have about the result of c—it cannot be the case that $c(x)$ "really" has the smaller datasort δ′—so the assumption $c(x) : \delta'(i)'$ is just as inconsistent as $1 \doteq 2$ is,

so in that case we need not examine e at all.

Remark 2.8. The second point may not be obvious. Suppose we have datasorts nonempty and list such that nonempty $\preceq$ list, but of course list $\not\preceq$ nonempty, and $\mathcal{S}(\mathsf{Nil}) = \mathbf{1} \to \mathsf{list}$. Given an e' : nonempty, to derive **case** e' **of** Nil $\Rightarrow e : C$ we try to derive

$$\Gamma; \mathsf{Nil} : \mathbf{1} \to \mathsf{list}; \mathsf{Nil}(x) : \mathsf{nonempty} \vdash e : C.$$

Since list $\not\preceq$ nonempty, we can apply rule δF-ct. The judgment expresses the assumptions that Nil has type $\mathbf{1} \to$ list and a particular value Nil(x) has type nonempty. If these assumptions were consistent, rule δF-ct would be wrong! But it cannot be the case that Nil(x) : nonempty. We assume Nil : $\mathbf{1} \to$ list, given to us by the user's constructor signature $\mathcal{S}$; we have no other information about

Nil. We cannot possibly show that a value of type list "really" also has type nonempty. Therefore the assumptions are inconsistent.

However, if $\delta \preceq \delta'$ (rule δS-ct), either $\delta' \preceq \delta$ (morally, $\delta' = \delta$) which clearly makes the subsorting assumptions consistent, or $\delta' \not\preceq \delta$ (morally, $\delta \prec \delta'$) in which case $c(x) : \delta'(i')$ is not very specific—some information about the value was lost between its construction and its deconstruction-by-**case**—but is still consistent. Either way, we must check $e : C$ assuming that $x{:}A$ and $i \doteq i'$ hold. The latter can make the context inconsistent: if $c : A \to \delta(0)$ and $c(x) : \delta(3)$, we can obtain $\Gamma, x{:}A, 0 \doteq 3 \vdash e : C$ through a rule contra:

$$\frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e \ \mathsf{ok}}{\Gamma \vdash e : A}\ \mathsf{contra}$$

The bodies of impossible case arms are thus skipped: by δF-ct if the datasort is impossible, by contra if the index is impossible. The remaining rules are for the types $\wedge$, Π, and $\supset$, and recapitulate—as left rules—the structure of the corresponding introduction rules. For example, $\wedge$-ct moves from the assumption $c : A_1^{con} \wedge A_2^{con}$, in the conclusion, to $c : A_1^{con}$ in the first premise and $c : A_2^{con}$ in the second, just as $\wedge$I moves from $\cdots : A_1 \wedge A_2$ in its conclusion to $\cdots : A_1$ and $\cdots : A_2$ in its premises.

$$\frac{\begin{array}{c}\Gamma; c : A_1^{con}; c(x) : B \vdash e : C \\ \Gamma; c : A_2^{con}; c(x) : B \vdash e : C\end{array}}{\Gamma; c : A_1^{con} \wedge A_2^{con}; c(x) : B \vdash e : C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : \Pi a{:}\gamma.\, A^{con}; c(x) : B \vdash e : C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : (P \supset A^{con}); c(x) : B \vdash e : C}\ \supset\text{-ct}$$

The subtyping rules for Π are

$$\frac{\Gamma \vdash [i/a]A \leq B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ \Pi\mathrm{L} \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ \Pi\mathrm{R}$$

The left rule allows one to instantiate a quantified index variable a to an index i of appropriate sort. The right rule states that if $A \leq B$ regardless of an index variable b, A is also a subtype of $\Pi b{:}\gamma.\, B$. Of course, b cannot occur free in A.

The typing rules for Π are

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\ \Pi\mathrm{I} \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\ \Pi\mathrm{E}$$

Like $\wedge$I, and for similar reasons (to maintain type preservation), ΠI is restricted to values. Moreover, if γ is an empty sort, progress would fail if the rule were not thus restricted.

2.3.5 Guarded types

The *guarded type* $P \supset A$, read "P implies A", is equivalent to A if the proposition P holds, and is useless otherwise (as $\top$ is useless). To illustrate, suppose our index domain is the integers (index sort $\mathcal{Z}$) with propositions $P ::= i < j \mid i \leq j \mid \ldots$. Then

$$\Pi a{:}\mathcal{Z}.\, (a \geq 0) \supset (\mathsf{int}(a) \to \mathsf{list}(a))$$

is the type of functions from natural numbers to lists of the corresponding length: a function of this type can only be applied to arguments $\mathsf{int}(i)$ such that $i \geq 0$.[3]

The introduction and elimination rules are quite natural. Note that to ensure progress the introduction rule—like the introduction rules for other definite types—is restricted to values: otherwise, we could use contra and $\supset$I to show $\vdash ()() : (\bot \supset A)$, which is not a value but does not take a step.

$$\frac{\Gamma, P \vdash v : A}{\Gamma \vdash v : P \supset A}\supset\mathrm{I} \qquad \frac{\Gamma \vdash e : P \supset A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : A}\supset\mathrm{E}$$

The subtyping rules for $\supset$ are straightforward: reading *both* $\supset$ and $\leq$ as "implies", the left rule $\supset$L says that if P holds, and A implies B, then (P implies A) implies B. Reading $\vdash$ as "implies" as well, the right rule $\supset$R says that if P implies (A implies B), then A implies (P implies B).

$$\frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash (P \supset A) \leq B}\supset\mathrm{L} \qquad \frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash A \leq (P \supset B)}\supset\mathrm{R}$$

Xi's early work [Xi00, Xi98] achieved the effect of $P \supset A$ through a different mechanism, the *subset sort* $\{a{:}\gamma \mid P\}$, which allows a constraint to be placed on the sort:

$$\Pi a{:}\overbrace{\{a{:}\mathcal{Z} \mid a \geq 0\}}.\, \mathsf{int}(a) \to \mathsf{list}(a)$$

Recent work by Xi [Xi04] uses a form of guarded types (in a somewhat different setting). Our type system does not include subset sorts; however, our implementation permits them through a preprocessing phase described in Section 6.3.4. For example, the *repeat* function's type, $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$, becomes $\Pi a{:}\mathcal{Z}.\, (a \geq 0) \supset (\mathsf{list}(a) \to \mathsf{even}(2 * a))$.

2.4 Indefinite property types

We now have a system with definite types $\wedge$, $\top$, Π, and $\supset$. The typing and subtyping rules are both orthogonal and internally regular: no rule mentions both $\top$ and $\wedge$, $\top$I is a 0-ary version of $\wedge$I, and so on. However, one cannot express the types of functions with indeterminate result type. A simple example is a *filter* $f\ l$ function on lists of integers, which returns the elements of l for which u returns true. It has the ordinary type

$$\textit{filter} : (\mathsf{int} {\to} \mathsf{bool}) \to \mathsf{list} \to \mathsf{list}$$

Indexing lists by their length, the refined type should look like[4]

$$\textit{filter} : (\mathsf{int} {\to} \mathsf{bool}) \to \Pi n{:}\mathcal{N}.\, \mathsf{list}(n) \to \mathsf{list}(__)$$

[3] In this and other examples, we use a singleton type $\mathsf{int}(i)$: every literal integer n has type $\mathsf{int}(n)$.

[4] In the past, we (and others) have given the refined type of *filter* with the Π on the outside, i.e. $\Pi n{:}\mathcal{N}.\, (\mathsf{int} {\to} \mathsf{bool}) \to \ldots$, which is quite unfortunate if *filter* is partially applied: n must be instantiated before *filter* is applied to its first argument, so in **let** *filter_f* = *filter* f **in** ..., the function *filter_f* can only be applied to lists known to all have the same length!

But we cannot fill in the blank. Xi's solution [XP99, Xi98] was to add dependent sums $\Sigma a{:}\gamma.\, A$ quantifying existentially over index variables. Then we can express the fact that *filter* returns a list of some indefinite length m as follows[5]:

$$\textit{filter} : (\mathsf{int}{\rightarrow}\mathsf{bool}) \rightarrow \Pi n{:}\mathcal{N}.\, \mathsf{list}(n) \rightarrow (\Sigma m{:}\mathcal{N}.\, \mathsf{list}(m))$$

For similar reasons, we also occasionally need 0-ary and binary indefinite types—the empty type and union types, respectively. We begin with the binary case.

2.4.1 Unions

On values, the binary indefinite type should simply be a union in the ordinary sense: if $\vdash v : A \vee B$ then either $\vdash v : A$ or $\vdash v : B$. This leads to the following subtyping rules, which are dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 {\vee} A_2 \leq B} \vee\mathrm{L} \qquad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 {\vee} B_2} \vee\mathrm{R}_1 \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 {\vee} B_2} \vee\mathrm{R}_2$$

The introduction rules directly express the simple logical interpretation:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B} \vee\mathrm{I}_1 \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B} \vee\mathrm{I}_2$$

The elimination rule is harder to formulate. It is clear that if $e : A \vee B$ and e evaluates to a value v, then either $v : A$ or $v : B$. So we should be able to reason by cases, similar to the usual disjunction elimination rule in natural deduction [Pra65, p. 20]. However, there are several complications. The first is that $A \vee B$ is a property type. That is, we cannot have a **case** construct in the ordinary sense since the members of the union are not tagged.[6]

As a simple example, consider

$$\begin{array}{rcl} f & : & (B \rightarrow D) \wedge (C \rightarrow D) \\ g & : & A \rightarrow (B \vee C) \\ x & : & A \end{array}$$

Then $f(g(x))$ should be type correct and have type D. At first this might seem doubtful, because the type of f does not directly show how to treat an argument of type $B \vee C$. However, whatever g returns must be a closed value v, and must therefore either have type B or type C. In both cases $f(v)$ should be well-typed and return a result of type D.

Note that we can distinguish cases on the result of $g(x)$ because it is evaluated *before* f is called.[7] In general, we allow case distinction on the type of the next expression to be evaluated. This guarantees both progress and preservation. The rule is then

[5]The additional constraint $m \leq n$ can be expressed by an *asserting type* (Section 2.4.4).

[6]Pierce's **case** [Pie91b] is a syntactic marker indicating where to apply the elimination rule. Clearly, a pure type assignment system should avoid this. We can avoid it even in a bidirectional system; see the rest of the thesis.

[7]If arguments were passed by name instead of by value, this would be unsound in a language with effects: evaluation of the same argument $e : A \vee B$ could sometimes return a value of type A and sometimes a value of type B. For example, let $e = e_1 \oplus e_2$ where $\oplus$ is a nondeterministic choice operator: both $e_1 \oplus e_2 \mapsto e_1$ and $e_1 \oplus e_2 \mapsto e_2$ are possible. Given $f(e)$ with $f = \lambda x.\, e'$, if x appears more than once in e', call by name evaluation would yield multiple copies of $e_1 \oplus e_2$,

$$\frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{l}\Gamma, x{:}A \vdash \mathcal{E}[x] : C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\Gamma \vdash \mathcal{E}[e'] : C} \vee\mathrm{E}$$

The use of the evaluation context $\mathcal{E}$ guarantees that e' is the next expression to be evaluated (or is some value), following our informal reasoning above. In the example, $e' = g(x)$ and $\mathcal{E} = f\,[\,]$.

Several generalizations of this rule come to mind that are in fact unsound in our setting. For example, allowing simultaneous parallel case distinction over several occurrences of e', as in a rule proposed in [BDCd95],

$$\frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{l}\Gamma, x{:}A \vdash e : C \\ \Gamma, x{:}B \vdash e : C\end{array}}{\Gamma \vdash [e'/x]\, e : C} \vee\mathrm{E}'$$

is unsound here: two occurrences of the identical e' could return different results (the first of type A, the second of type B), while the rule above assumes consistency. Similarly, we cannot allow the occurrence of e' to be in a position where it might not be evaluated. That is, in $\vee\mathrm{E}'$ it is not enough to require that there be exactly one occurrence of x in e, because, for example, if we consider the context

$$\begin{array}{lcl} f & : & ((1 \to B) \to D) \wedge ((1 \to C) \to D), \\ g & : & A \to (B \vee C), \\ x & : & A \end{array}$$

and term $f\,(\lambda y.\, g(x))$, then f may use its argument at multiple types, eventually evaluating $g(x)$ multiple times with different possible answers. Thus, treating it as if all occurrences must all have type B or all have type C is unsound. If we restrict the rule so that e' must be a value, as in [vBDCdM00], we obtain a sound but impractical rule—a typechecker would have to guess e' and, if it occurs more than once, a subset of its occurrences.

A final generalization suggests itself: we might allow the subterm e' to occur exactly once, and in any position where it would definitely have to be evaluated exactly once for the whole expression to be evaluated. Besides the difficulty of characterizing such positions, even this apparently innocuous generalization is unsound for the empty type $\bot$, as discussed in the next section.

2.4.2 The empty type

The 0-ary indefinite type is the empty or void type $\bot$; it has no values. For $\top$ we had one right subtyping rule; for $\bot$, following the principle of duality, we have one left rule:

$$\frac{}{\Gamma \vdash \bot \leq A} \bot\mathrm{L}$$

For example, the term $\omega = (\mathbf{fix}\ u.\, \lambda x.\, u(x))()$ has type $\bot$. For an elimination rule $\bot\mathrm{E}$, we can

---

which would not all have to reduce in the same way. Suppose $e' = g\ x\ x$ where $g : (A \to A \to C) \wedge (B \to B \to C)$; then $f(e) \mapsto^* g\ (e_1 \oplus e_2)\ (e_1 \oplus e_2) \mapsto g\ e_2\ (e_1 \oplus e_2) \mapsto g\ e_2\ e_1$, which is ill typed.

Mutable references also serve: suppose $e = \,!r$ where $r : (\mathsf{true} \vee \mathsf{false})$ **ref**. With the appropriate refinement of type bool, we can give e the type $\mathsf{true} \vee \mathsf{false}$. However, occurrences of e will variously reduce to values of either type true or false, depending on intervening assignments to r.

proceed by analogy with $\vee$E:

$$\frac{\Gamma \vdash e' : \bot \qquad \Gamma \vdash \mathcal{E}[e'] \ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] : C} \ \bot\mathrm{E}$$

As before, the expression typed must be an evaluation context $\mathcal{E}$ with redex e'. Viewing $\bot$ as a 0-ary union, we had two additional premises in $\vee$E, so we have none now. $\bot$E is sound, but the generalization mentioned at the end of the previous section would violate progress (Theorem 2.21). This is easy to see through the counterexample $(()\,())\omega$.

2.4.3 Existential dependent types: Σ

Now we add an infinitary indefinite type Σ. Just as we have come to expect, the subtyping rules are dual to the rules for the corresponding definite type (in this case Π):

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B} \ \Sigma\mathrm{L} \qquad \frac{\Gamma \vdash A \leq [i/b]\, B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B} \ \Sigma\mathrm{R}$$

The typing rule that introduces Σ is simply

$$\frac{\Gamma \vdash e : [i/a]\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\, A} \ \Sigma\mathrm{I}$$

For the elimination rule, we again have the restriction to evaluation contexts:

$$\frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C} \ \Sigma\mathrm{E}$$

Not only is the restriction consistent with the elimination rules for $\bot$ and $\vee$, but it is required. The counterexample for $\bot$ suffices: Suppose that the rule were unrestricted, so that it typed any e containing some subterm e'. Let $e' = \omega$ and $e = (()\,())(\omega)$. Since $e' : \bot$, by subsumption e' has type $\Sigma a{:}\bot.\, A$ for any A, and by the contra rule, $a{:}\bot, x{:}A \vdash (()\,())x : C$ (where $\bot$ is the empty sort). Now we can apply the unrestricted rule to conclude $\vdash (()\,())e' : C$ for any C, contrary to progress.

2.4.4 Asserting types

The *asserting type* $P \otimes A$, read "P with A", is as the type A, but also asserts that P holds. To illustrate, suppose our index domain is the integers (index sort $\mathcal{Z}$) with propositions $P ::= i < j \mid i \leq j \mid \ldots$. Then

$$\mathbf{1} \to \Sigma a{:}\mathcal{Z}.\, ((a > 0) \otimes \mathsf{int}(a))$$

is the type of functions from unit to strictly positive integers.

On a high level, following Curry-Howard, we can think of a term of type $P \otimes A$ as a proof of both P and A, where only A has computational content. On an even higher level, we can think of $P \otimes A$ as simply another kind of conjunction that conjoins a proposition and a type, rather than two types as $\wedge$ does. Following this intuition, the subtyping rules and introduction rule are straightforward.

$$\frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash (P \otimes A) \leq B}\ \otimes\text{L} \qquad \frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq (P \otimes B)}\ \otimes\text{R}$$

$$\frac{\Gamma \vdash e : A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : P \otimes A}\ \otimes\text{I}$$

The elimination rule, however, follows the pattern of ΣE.

$$\frac{\Gamma \vdash e' : P \otimes A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \otimes\text{E}$$

The subset sort mechanism, discussed above in the context of asserting types, can achieve the effect of $P \otimes A$:

$$\mathbf{1} \to \Sigma a{:}\overbrace{\{a{:}\mathcal{Z} \mid a > 0\}}.\,\mathsf{int}(a) \qquad\qquad \textit{(Not in our system!)}$$

We consider $\otimes$ to be an indefinite type. It may seem strange to have a type that looks like a conjunction in the same category as $\vee$ and Σ, which appear disjunctive. Consider that the *definite* types are closely linked to the allowed constructor types A^{con}: a constructor type is built up from intersections and guards of arrows $B \to \delta(i)$. The indefinite types, on the other hand, cannot be permitted in the codomain of a constructor type: if $\mathcal{S}(c) = B \to \bot$, we could construct values of the empty type! Likewise, if $\mathcal{S}(c) = B \to ((2 \doteq 3) \otimes \delta(i))$, we can construct a value of type $(2 \doteq 3) \otimes \delta(i)$, which asserts that $2 \doteq 3$. The fact that constructor types are as hostile to $\otimes$ as to $\bot$ strongly suggests that $\otimes$ is indefinite.[8]

The structure of the type system also suggests that $\otimes$ belongs with $\vee$ and Σ. To take just one example, $\otimes$L adds to the context in its premise, as ΣL does—and as ΠL does not.

2.4.5 Typechecking in evaluation order

The following rule internalizes a kind of substitution principle for evaluation contexts and allows us to check a term in evaluation order.

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\ \mathsf{direct}$$

Perhaps surprisingly, this rule is not only admissible but derivable in our system: from $e' : A$ we can conclude $e' : A \vee A$ and then apply $\vee$E. However, the corresponding bidirectional rule is not admissible, and so must be primitive in a bidirectional system (Chapter 3).

Thus, in either the type assignment or bidirectional systems, we can choose to typecheck the term in evaluation order. This has a clear parallel in Xi's work [Xi98], which is bidirectional and contains both Π and Σ. There, the order in which terms are typed is traditional, not guided by evaluation order. However, Xi's elaboration algorithm in the presence of Π and Σ transforms the term into a let-normal form, which has a similar effect. We return to this point in Chapter 5.

$$\begin{array}{lrl} \text{Types} & A, B, C, D ::= & \mathbf{1} \mid A \to B \mid A * B \mid \delta(i) \mid A \wedge B \mid \top \mid \Pi a{:}\gamma.\, A \mid P \supset A \\ & & \mid A \vee B \mid \bot \mid \Sigma a{:}\gamma.\, A \mid P \parr A \end{array}$$

$\boxed{\Gamma \vdash e : A}$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\,\textsf{var} \qquad \frac{\Gamma(u) = A}{\Gamma \vdash u : A}\,\textsf{fixvar} \qquad \frac{\Gamma, u{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\; u.\, e : A}\,\textsf{fix}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\,\textsf{sub}$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\,{\to}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\,{\to}\mathrm{E}$$

$$\frac{}{\Gamma \vdash () : \mathbf{1}}\,\mathbf{1}\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 * A_2}\,{*}\mathrm{I} \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{fst}(e) : A}\,{*}\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : A * B}{\Gamma \vdash \mathbf{snd}(e) : B}\,{*}\mathrm{E}_2$$

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \qquad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\,\delta\mathrm{I} \qquad \frac{\Gamma \vdash e : \delta(i) \qquad \Gamma \vdash ms :_{\delta(i)} C}{\Gamma \vdash \mathbf{case}\; e\; \mathbf{of}\; ms : C}\,\delta\mathrm{E} \qquad \frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e\;\textsf{ok}}{\Gamma \vdash e : A}\,\textsf{contra}$$

$$\frac{\Gamma \vdash v : A_1 \qquad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}\,{\wedge}\mathrm{I} \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\,{\wedge}\mathrm{E}_1 \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\,{\wedge}\mathrm{E}_2$$

$$\frac{\Gamma \vdash v\;\textsf{ok}}{\Gamma \vdash v : \top}\,\top\mathrm{I}$$

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\,\Pi\mathrm{I} \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\,\Pi\mathrm{E}$$

$$\frac{\Gamma, P \vdash v : A}{\Gamma \vdash v : P \supset A}\,{\supset}\mathrm{I} \qquad \frac{\Gamma \vdash e : P \supset A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : A}\,{\supset}\mathrm{E}$$

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B}\,{\vee}\mathrm{I}_1 \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B}\,{\vee}\mathrm{I}_2 \qquad \frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] : C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\Gamma \vdash \mathcal{E}[e'] : C}\,{\vee}\mathrm{E}$$

$$\frac{\Gamma \vdash e' : \bot \qquad \Gamma \vdash \mathcal{E}[e']\;\textsf{ok}}{\Gamma \vdash \mathcal{E}[e'] : C}\,\bot\mathrm{E}$$

$$\frac{\Gamma \vdash e : [i/a]\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\, A}\,\Sigma\mathrm{I} \qquad \frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\,\Sigma\mathrm{E}$$

$$\frac{\Gamma \vdash e : A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e : P \parr A}\,{\parr}\mathrm{I} \qquad \frac{\Gamma \vdash e' : P \parr A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\,{\parr}\mathrm{E}$$

$$\frac{\Gamma \vdash e' : A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] : C}{\Gamma \vdash \mathcal{E}[e'] : C}\,\textsf{direct}$$

Figure 2.9: Typing rules

$$\boxed{\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e : C}$$

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i' \vdash e : C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta' \qquad \Gamma, x{:}A \vdash e \text{ ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e : C}\ \delta\text{F-ct}$$

$$\frac{\begin{array}{c}\Gamma; c : A_1^{con}; c(x) : B \vdash e : C \\ \Gamma; c : A_2^{con}; c(x) : B \vdash e : C\end{array}}{\Gamma; c : A_1^{con} \wedge A_2^{con}; c(x) : B \vdash e : C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : \Pi a{:}\gamma.\, A^{con}; c(x) : B \vdash e : C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A^{con}; c(x) : B \vdash e : C}{\Gamma; c : (P \supset A^{con}); c(x) : B \vdash e : C}\ \supset\text{-ct}$$

$$\boxed{\Gamma \vdash ms :_B C}$$

$$\frac{}{\Gamma \vdash \cdot :_B C}\ \text{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e : C \qquad \Gamma \vdash ms :_B C}{\Gamma \vdash (c(x) \Rightarrow e \mid ms) :_B C}\ \text{casearm}$$

Figure 2.10: Case typing rules

$$\boxed{\Gamma \vdash A \leq B}$$

$$\frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ \to \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ \mathbf{1} \qquad \frac{\Gamma \vdash A_1 \leq B_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 * A_2 \leq B_1 * B_2}\ *$$

$$\frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ \wedge R \qquad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ \wedge L_1 \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ \wedge L_2$$

$$\frac{\delta_1 \preceq \delta_2 \qquad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ \delta \qquad \frac{\Gamma \vdash [i/a]A \leq B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\ \Pi L \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\ \Pi R$$

$$\frac{\Gamma \vdash A_1 \leq B \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B}\ \vee L \qquad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2}\ \vee R_1 \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2}\ \vee R_2$$

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B}\ \Sigma L \qquad \frac{\Gamma \vdash A \leq [i/b]\, B \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B}\ \Sigma R \qquad \frac{}{\Gamma \vdash \bot \leq A}\ \bot L \qquad \frac{}{\Gamma \vdash A \leq \top}\ \top R$$

$$\frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash (P \supset A) \leq B}\ \supset L \qquad \frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash A \leq (P \supset B)}\ \supset R$$

$$\frac{\Gamma, P \vdash A \leq B}{\Gamma \vdash (P \otimes A) \leq B}\ \otimes L \qquad \frac{\overline{\Gamma} \models P \qquad \Gamma \vdash A \leq B}{\Gamma \vdash A \leq (P \otimes B)}\ \otimes R$$

Figure 2.11: Subtyping rules

2.5 Properties of subtyping

We assume that $\models$ and $\vdash$ in the constraint domain are decidable. We also assume that we can somehow guess the index i in rules ΠL and ΣR (in practice, this requires introducing an existential variable and solving for it; see Chapter 6). Under these assumptions, we can show:

Theorem. $\Gamma \vdash A \leq B$ *is decidable.*

Proof. Straightforward, since for each subtyping rule (Figure 2.11), every premise is smaller than the conclusion. □

We omitted rules for reflexivity and transitivity of subtyping without loss of expressive power, because they are admissible:

Lemma 2.9 (Reflexivity and Transitivity of $\leq$). *For any context* Γ*:*

(i) $\Gamma \vdash A \leq A$*;*

(ii) if $\Gamma \vdash A \leq B$ *and* $\Gamma \vdash B \leq C$ *then* $\Gamma \vdash A \leq C$.

Proof. For (i), by induction on A. For (ii), by induction on the given subtyping derivations; in each case at least one derivation becomes smaller. In the cases ΣR/ΣL and ΠR/ΠL we substitute an index i for a parameter a in a derivation. □

In addition we have a large set of inversion properties, which are purely syntactic in our system. We elide the lengthy statement of these properties here.

Note that we do not pretend to any completeness of subtyping with respect to the value interpretation (or anything else): the fact that the values of A are a subset of the values of B does not guarantee $A \leq B$. For example, both $\bot * \bot$ and $\bot$ are uninhabited, but $\bot * \bot \leq \bot$ cannot be derived.

2.6 Properties of values

For the proof of type safety, we need a key property: *values are always definite*. That is, once we obtain a value v, even though v might have type $A \vee B$, it *must* be possible to assign a definite type to v. In order to make this precise, we formulate substitutions σ that substitute values and indices, respectively, for several program variables x and index variables a. First we prove a simple lemma relating values and evaluation contexts.

Lemma 2.10. *If* $\mathcal{E}[e']$ value *then: (1)* e' value*; (2) for any* v value*,* $\mathcal{E}[v]$ value.

Proof. By structural induction on $\mathcal{E}$. □

$\boxed{\Gamma' \vdash \sigma : \Gamma}$

$$\frac{}{\Gamma' \vdash \cdot : \cdot}\ \text{empty-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \overline{\Gamma'} \models [\sigma]P}{\Gamma' \vdash \sigma : (\Gamma, P)}\ \text{prop-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \overline{\Gamma'} \vdash i : \gamma}{\Gamma' \vdash (\sigma, i/a) : (\Gamma, a{:}\gamma)}\ \text{ivar-}\sigma$$

$$\frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash v : [\sigma]A}{\Gamma' \vdash (\sigma, v/x) : (\Gamma, x{:}A)}\ \text{pvar-}\sigma \qquad \frac{\Gamma' \vdash \sigma : \Gamma \qquad \Gamma' \vdash u : [\sigma]A}{\Gamma' \vdash \sigma : (\Gamma, u{:}A)}\ \text{fixvar-}\sigma$$

Figure 2.12: Substitution typing

2.6.1 Substitutions

Figure 2.12 defines a typing judgment for substitutions $\Gamma' \vdash \sigma : \Gamma$. It could be more general; here we are only interested in substitutions of values for program variables and indices for index variables that verify the logical assumptions of the constraint domain. Note in particular that substitutions σ do not substitute for fixed point variables, though rule fixvar-σ allows them to appear in the contexts; for example, one can derive $b{:}\mathcal{Z}, u{:}\mathsf{list}(b)\to\mathbf{1} \vdash b/a : (a{:}\mathcal{Z}, u{:}\mathsf{list}(b)\to\mathbf{1})$. Application of a substitution σ to a term e or type A is in the usual (capture-avoiding) manner.

Definition 2.11. The *identity substitution* Γ/Γ is defined thus:

$$\begin{aligned} \cdot/\cdot &= \cdot \\ (\Gamma, x{:}A)/(\Gamma, x{:}A) &= (\Gamma/\Gamma), x/x \\ (\Gamma, a{:}\gamma)/(\Gamma, a{:}\gamma) &= (\Gamma/\Gamma), a/a \\ (\Gamma, P)/(\Gamma, P) &= \Gamma/\Gamma \\ (\Gamma, u{:}A)/(\Gamma, u{:}A) &= \Gamma/\Gamma \end{aligned}$$

Proposition 2.12 (Identity Substitution Typing)**.** *For all* Γ *such that* Γ/Γ *is defined,* $\Gamma, \Gamma' \vdash \Gamma/\Gamma : \Gamma$.

Proof. By induction on Γ, using the easily proved identities $[\Gamma/\Gamma]\,A = A$, $[\Gamma/\Gamma]\,i = i$, $[\Gamma/\Gamma]\,P = P$. □

Lemma 2.13 (Weakening)**.**

(i) If $\Gamma \vdash B \leq C$ *then* $\Gamma, x{:}A \vdash B \leq C$.

(ii) If $\Gamma \vdash e : C$ *then* $\Gamma, x{:}A \vdash e : C$.

(iii) If $\Gamma' \vdash \sigma : \Gamma$ *then* $\Gamma', x{:}A \vdash \sigma : \Gamma$.

(And similarly for $\Gamma \vdash ms :_B C$ *and* $\Gamma; c : B^{con}; c(x) : \delta(i) \vdash e : C$.*)*

Proof. By induction on the given derivation. We show part (iii): For empty-σ the result is immediate; for ivar-σ and prop-σ, $\overline{\Gamma', x{:}A} = \overline{\Gamma'}$. For pvar-$\sigma$ and fixvar-σ, use the IH (ii). □

Lemma 2.14 (Substitution)**.** *(i) If* $\Gamma \vdash A \leq B$ *and* $\Gamma' \vdash \sigma : \Gamma$*, then* $\Gamma' \vdash [\sigma]A \leq [\sigma]B$.

(ii) If $\mathcal{D} :: \Gamma \vdash e : A$ and $\Gamma' \vdash \sigma : \Gamma$, then there exists $\mathcal{D}' :: \Gamma' \vdash [\sigma]e : [\sigma]A$. Moreover, if $\overline{\Gamma} = \Gamma$ (that is, Γ contains only index variables and index constraints), there is such a $\mathcal{D}'$ not larger than $\mathcal{D}$ (that is, there are no more typing rule applications in $\mathcal{D}'$ than in $\mathcal{D}$).

"$\mathcal{D}$ ::" is read "$\mathcal{D}$ derives" (Sec. 1.9.2).

(iii) If $\Gamma \vdash e' : B$ and $\Gamma, u{:}B \vdash e : A$ then $\Gamma \vdash [e'/u]\, e : A$.

(And similar properties hold for $\Gamma \vdash ms :_C B$ and $\Gamma; c : C^{con}; c(x) : \delta(i) \vdash e : A$.)

Proof. By induction on the respective derivations. □

2.6.2 Definiteness

A typing judgment is *definite* if, whenever it shows that a term has indefinite type, we can also show that the term has a more particular type. For example, the judgment $\cdot \vdash \mathsf{Cons}(0, \mathsf{Nil}) : \mathsf{list}(0) \vee \mathsf{list}(1)$ is definite because we can also show $\cdot \vdash \mathsf{Cons}(0, \mathsf{Nil}) : \mathsf{list}(1)$. In this section, we show that every judgment typing a closed value is definite. More specifically, we show that

(i) $\vdash v : \bot$ is *not* derivable;

(ii) if $\vdash v : A \vee B$ then $\vdash v : A$ or $\vdash v : B$;

(iii) if $\vdash v : \Sigma a{:}\gamma.\, A$ then $\vdash v : [i/a]A$ for some i;

(iv) if $\vdash v : P \otimes A$ then $\vdash v : A$ and $\models P$.

For the proof of definiteness (and the later proofs of value inversion and type safety), we use a nontrivial measure of derivation size. This measure is motivated by the premises of the indefinite elimination rules such as $\vee$E, where we must substitute for a variable in the proof cases for the indefinite elimination rules, such as $\vee$E. That rule includes a premise introducing a new variable x. We will have a value e' to substitute for x, but if we use only a simple measure of derivation size, we would be unable to apply the IH to the resulting derivation (which is required) because it could become larger. We obtain a valid inductive proof by stratifying derivations into those of *rank* 0, 1, etc., where the rank of $\mathcal{D}$ is (roughly) the number of applications of indefinite elimination rules in $\mathcal{D}$. We will ensure that the derivations $\mathcal{D}'$ *produced* by the theorem have rank 0. Thus, applying the IH to the typing for the value e', and substituting e' for x, cannot increase rank because the substituted derivations typing e' have rank 0—and the rank of a subderivation of any premise of $\vee$E is one less than the rank of the original derivation, because that original derivation ends in $\vee$E.

Of course, we also want to apply the IH to premises of other rules, say $\wedge$I, where the rank of the derivation of the premises is no smaller, so we use a lexicographical ordering of the rank (notated *Rank*) and the usual measure of derivation size. We also define the rank of any derivation concluding in certain rules to be 0, regardless of applications of $\vee$E, etc. in its subderivations. This lets us handle cases like $e' = \lambda x.\, e_1$ where the typing of e_1 itself uses $\vee$E. Substituting a typing derivation of e' into another derivation $\mathcal{D}_1$ may increase the number of times $\vee$E is used in $\mathcal{D}_1$,

[8] Σ with an empty index sort $\bot$ can also construct an abomination, if $\mathcal{S}(c) = B \to \Sigma a{:}\bot.\, \delta(a)$. Allowing unions in constructor codomains is not immediately catastrophic, but would break our key property of value definiteness (below).

for example, if $\mathcal{D}_1$ uses $\wedge$I. But we can ignore those uses since they are “buried” inside the typing derivation of e_1, and the proof never penetrates the derivation concluding in $\to$I.

An illustration of this notion is given in Figure 2.13. Rank depends only on the rules applied and not the judgments, so we elide the judgments, showing ranks in their place. At the root of the derivation, direct is applied to subderivations of ranks 0 and 1, resulting in a derivation of rank 2. The right-hand subderivation concludes in an application of $\wedge$I; its rank is just the sum of its subderivations' ranks $(1 + 0 = 1)$. Moving up, we have a $\vee$E application to three rank 0 derivations, yielding a rank 1 derivation. However, the ΣE application at the upper right, which yields a derivation of rank 1, does not add to the rank of the larger derivation because there is an intervening $\to$I, forcing the rank to be 0.

$$\dfrac{\mathbf{0} \qquad \dfrac{\dfrac{\mathbf{0}\quad\mathbf{0}\quad\mathbf{0}}{\mathbf{1}}\,\vee\mathrm{E} \qquad \dfrac{\dfrac{\mathbf{0}\quad\mathbf{0}}{\mathbf{1}}\,\Sigma\mathrm{E}}{\mathbf{0}}\,\to\mathrm{I}}{\mathbf{1}}\,\wedge\mathrm{I}}{\mathbf{2}}\,\mathsf{direct}$$

Figure 2.13: Illustration of derivation rank

Definition 2.15 (Definiteness Derivation Measure)**.** The measure μ of a derivation $\mathcal{D}$ is a lexicographically ordered pair

$$\mu(\mathcal{D}) = \langle \mathit{Rank}(\mathcal{D}), \mathit{Size}(\mathcal{D})\rangle$$

Let $\mathcal{R}$ be the rule concluding $\mathcal{D}$, and let $\mathcal{D}_1, \dots, \mathcal{D}_n$ be the n subderivations, one for each premise of $\mathcal{R}$ deriving a typing judgment. Then $\mathit{Size}(\mathcal{D})$ is just the simple derivation size, that is,

$$\mathit{Size}(\mathcal{D}) = 1 + \mathit{Size}(\mathcal{D}_1) + \cdots + \mathit{Size}(\mathcal{D}_n)$$

and $\mathit{Rank}(\mathcal{D})$ is defined as follows.

$$\mathit{Rank}(\mathcal{D}) = \begin{cases} 0 & \text{if } \mathcal{R} \text{ is } \mathsf{fix} \text{ or } \to\mathrm{I} \\ 1 + \mathit{Rank}(\mathcal{D}_1) + \cdots + \mathit{Rank}(\mathcal{D}_n) & \text{if } \mathcal{R} \text{ is } \bot\mathrm{E}, \vee\mathrm{E}, \Sigma\mathrm{E}, \invamp\mathrm{E} \text{ or } \mathsf{direct} \\ \mathit{Rank}(\mathcal{D}_1) + \cdots + \mathit{Rank}(\mathcal{D}_n) & \text{otherwise} \end{cases}$$

We also define $\mathit{Rank}(\sigma)$ of a substitution $\Gamma' \vdash \sigma : \Gamma$ as the sum of the ranks of its constituent typing derivations, slightly abusing terminology since it is really the *derivation* of the substitution typing judgment that has rank. Thus, if $\sigma = e'/x$ has rank 0, the derivation of $\Gamma' \vdash e' : A$ must also have rank 0.

We also need a “ranked” version of the substitution lemma (2.14) that says, roughly, that substituting rank 0 derivations for rank 0 derivations (specifically, applications of var) cannot increase rank; therefore, applying a rank 0 substitution does not increase the rank of the resulting typing derivation. For consistency with the earlier lemma, we designate the whole thing “(ii)”.

Lemma 2.16 (Ranked Substitution).

(ii) *If* $\mathcal{D} :: \Gamma \vdash e : A$ *and* $\cdot \vdash \sigma : \Gamma$*, where* $Rank(\sigma) = 0$ *then there exists* $\mathcal{D}' :: \cdot \vdash [\sigma]e : [\sigma]A$*, where* $Rank(\mathcal{D}') = Rank(\mathcal{D})$.

Moreover, if $\overline{\Gamma} = \Gamma$ *it is the case that* $Size(\mathcal{D}') \leq Size(\mathcal{D})$.

Proof. By induction on the derivation $\mathcal{D}$.

In the fix and $\rightarrow$I cases, we apply Lemma 2.14 on each premise, then apply the same rule, yielding a derivation that is rank 0 (according to Definition 2.15). Since $Rank(\mathcal{D})$ is also 0, we have the result.

In the var case, e is some variable x. Within the derivation of $\cdot \vdash \sigma : \Gamma$ there must lie a derivation $\mathcal{D}' :: \cdot \vdash e' : A$, where $[\sigma]x = e'$ and $\Gamma(x) = A$. We have $Rank(\sigma) = 0$, so $Rank(\mathcal{D}') = 0$, which was to be shown.

In all other cases, we simply apply the IH to each premise and apply the same rule. (For the rules that themselves add to the rank, such as $\vee$E, we have a derivation of rank n whose subderivations have ranks summing to $n - 1$. Applying the IH to each yields a derivation of the same rank. Therefore, applying $\vee$E to those derivations results in a derivation of rank $(n-1)+1 = n$.) For rules in which some premises are not typing judgments, such as sub and δE, we use Lemma 2.14 on them.

The "moreover" part follows from the "moreover" part of Lemma 2.14. □

Theorem 2.17 (Value Definiteness). *If* $\mathcal{D} :: \cdot \vdash v : A$ *then:*

(i) $\cdot \vdash A \leq \bot$ *is* not *derivable;*

(ii) *if* $\cdot \vdash A \leq B_1 \vee B_2$ *then there exists* $\mathcal{D}' :: \cdot \vdash v : B_k$ *for some* $k \in \{1, 2\}$*;*

(iii) *if* $\cdot \vdash A \leq \Sigma b{:}\gamma.\, B$ *then there exists* $\vdash i : \gamma$ *such that* $\mathcal{D}' :: \cdot \vdash A' \leq [i/b]B$*;*

(iv) *if* $\cdot \vdash A \leq P \otimes B$ *then* $\models P$ *and* $\mathcal{D}' :: \cdot \vdash v : B$*;*

(v) $\mathcal{D}' :: \cdot \vdash v : A$.[9]

Moreover, in parts (ii)–(v), $Rank(\mathcal{D}') = 0$.

Section 1.9.2 explains some of the notation used in our proofs.

Proof. By induction on $\mu(\mathcal{D})$, where $\mathcal{D} :: \cdot \vdash v : A$.

The term v is a value, so we need not consider rules that cannot type values. Furthermore, by Property 2.5 the contra case cannot arise. Most cases follow easily from the IH, reflexivity/-transitivity of subtyping, and rule sub. For example, in the case for ΠI, part (ii), we are given $\cdot \vdash \Pi a{:}\gamma.\, A_0 \leq B_1 \vee B_2$. The only rules that can derive such a judgment are ΠL, $\vee R_1$ and $\vee R_2$. If ΠL was used, use Lemma 2.16 to eliminate a in the premise of ΠI and apply the IH. If $\vee R_1$ was used, we have $\cdot \vdash \Pi a{:}\gamma.\, A_0 \leq B_1$, whence we can apply the IH. The $\vee R_2$ subcase is similar.

Rule var is impossible since the context is empty.

For sub, use transitivity of subtyping followed by the IH.

That leaves the contextual rules $\bot$E, $\vee$E, ΣE, $\otimes$E and direct; we show the first three cases (the last two are similar to $\vee$E, but using IH(iv) and IH(v), respectively):

[9] The force of part (v) comes from the "Moreover, ..." statement about $Rank(\mathcal{D}')$.

- **Case** $\bot$E: $$\mathcal{D} :: \dfrac{\cdot \vdash e' : \bot \qquad \cdot \vdash \mathcal{E}[e']\ \mathsf{ok}}{\cdot \vdash \mathcal{E}[e'] : A}$$

  By assumption, $\mathcal{E}[e']$ is a value. By Lemma 2.10, e' is a value. $\cdot \vdash \bot \leq \bot$ (Lemma 2.9), so by the IH (i), this case cannot arise.

- **Case** $\wedge$I: $$\mathcal{D} :: \dfrac{\cdot \vdash v : A_1 \qquad \cdot \vdash v : A_2}{\cdot \vdash v : A_1 \wedge A_2}$$

  Part (ii): It is given that $\vdash A_1 \wedge A_2 \leq B_1 \vee B_2$. The only rules that can derive such a judgment are $\vee R_{1,2}$ and $\wedge L_{1,2}$.

  If by $\vee R_1$, we have $\vdash A_1 \wedge A_2 \leq B_1$. Applying the IH (v) to each subderivation of $\mathcal{D}$ yields rank-0 derivations of $\vdash v : A_1$ and $\vdash v : A_2$; applying $\wedge$I to those yields a rank-0 derivation of $\vdash v : A_1 \wedge A_2$. By sub, $\vdash v : B_1$, which was to be shown. The $\vee R_2$ case is similar.

  If by $\wedge L_1$,we have $\vdash A_1 \leq B_1 \vee B_2$; the result follows by IH on $\cdot \vdash v : A_1$. The $\wedge L_2$ case is similar.

  The proofs of parts (i) and (iii)–(v) are similar.

- **Case** ΠI, $\supset$I, $\top$I: Similar to $\wedge$I.

- **Case** $\wedge E_1$: $$\mathcal{D} :: \dfrac{\cdot \vdash v : A \wedge A_0}{\cdot \vdash v : A}$$

  Part (ii): We have $\vdash A \leq B_1 \vee B_2$. By $\wedge L_1$ and transitivity (Lemma 2.9), $\cdot \vdash A \wedge A_0 \leq B_1 \vee B_2$. The result follows by IH on $\cdot \vdash v : A \wedge A_0$.

  The proofs of parts (i) and (iii)–(v) are similar.

- **Case** $\wedge E_2$, ΠE, $\supset$E: Similar to $\wedge E_1$.

- **Case** $\vee$E: $$\mathcal{D} :: \dfrac{\cdot \vdash e' : C_1 \vee C_2 \qquad \begin{matrix}\mathcal{D}_1 \\ x{:}C_1 \vdash \mathcal{E}[x] : A\end{matrix} \qquad \begin{matrix}\mathcal{D}_2 \\ y{:}C_2 \vdash \mathcal{E}[y] : A\end{matrix}}{\cdot \vdash \mathcal{E}[e'] : A}$$

  This case goes the same way for all parts, (i)–(v).

  $\mathcal{E}[e']$ value is given. By Lemma 2.10, $\mathcal{E}[x]$ value and e' value.

  By Lemma 2.9, $\vdash C_1 \vee C_2 \leq C_1 \vee C_2$. By IH (ii), $\mathcal{D}' :: \cdot \vdash e' : C_k$ for some $k \in \{1, 2\}$, where $\mathit{Rank}(\mathcal{D}') = 0$. Assume $k = 1$; the $k = 2$ case is similar. By empty-σ and pvar-σ, $\cdot \vdash e'/x : x{:}C_1$, which, since $\mathit{Rank}(\mathcal{D}') = 0$, is a rank-0 substitution. By Lemma 2.16 on $\mathcal{D}_1 :: x{:}C_1 \vdash \mathcal{E}[x] : A$, we have
  $$\mathcal{D}'' :: \cdot \vdash [e'/x]\ \mathcal{E}[x] : [e'/x]A$$
  where $\mathit{Rank}(\mathcal{D}'') = \mathit{Rank}(\mathcal{D}_1)$. Since x is new, it does not appear in $\mathcal{E}$, so in fact we have $\mathcal{D}'' :: \cdot \vdash \mathcal{E}[e'] : A$. By Definition 2.15, $\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$. Therefore $\mathit{Rank}(\mathcal{D}'') < \mathit{Rank}(\mathcal{D})$.

Under the lexicographic ordering, $\mu(\mathcal{D}'') < \mu(\mathcal{D})$, so we can apply the IH to $\mathcal{D}'' :: \cdot \vdash \mathcal{E}[e'] : A$, yielding the result.

This style of line-by-line proof is explained in Section 1.9.2.

- **Case** direct: $\mathcal{D} :: \dfrac{\cdot \vdash e' : C \qquad \overset{\mathcal{D}_1}{x{:}C \vdash \mathcal{E}[x] : A}}{\cdot \vdash \mathcal{E}[e'] : A}$

$\cdot \vdash e' : C$	Subderivation
$\mathcal{D}'_0 :: \cdot \vdash e' : C$ and $\mathit{Rank}(\mathcal{D}'_0) = 0$	By IH (v)
$\cdot \vdash e'/x : x{:}C$	By empty-σ then pvar-σ
$\mathcal{D}_1 :: x{:}C \vdash \mathcal{E}[x] : A$	Subderivation
$\mathcal{D}' :: \cdot \vdash [e'/x]\,\mathcal{E}[x] : A$	By Lemma 2.16
$\mathit{Rank}(\mathcal{D}') = \mathit{Rank}(\mathcal{D}_1)$	By Lemma 2.16
$\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$	By Definition 2.15
$[e'/x]\,\mathcal{E}[x]$ satisfies (i)–(v)	By IH
$[e'/x]\,\mathcal{E}[x] = \mathcal{E}[e']$	By defn. of substitution (x new)
☞ $\mathcal{E}[e']$ satisfies (i)–(v)	By previous equation

- **Case** $\otimes$E: $\mathcal{D} :: \dfrac{\cdot \vdash e' : P \otimes C \qquad \overset{\mathcal{D}_1}{P, x{:}C \vdash \mathcal{E}[x] : A}}{\cdot \vdash \mathcal{E}[e'] : A}$

$\cdot \vdash e' : P \otimes C$	Subderivation
$\models P$	By IH (iv)
$\mathcal{D}'_0 :: \cdot \vdash e' : C$ and $\mathit{Rank}(\mathcal{D}'_0) = 0$	By IH (iv)
$\vdash e'/x : x{:}C$	By empty-σ then pvar-σ
$\vdash e'/x : x{:}C, P$	By prop-σ
$\mathcal{D}_1 :: x{:}C, P \vdash \mathcal{E}[x] : A$	Subderivation
$\mathcal{D}' :: \cdot \vdash [e'/x]\,\mathcal{E}[x] : A$	By Lemma 2.16
$\mathit{Rank}(\mathcal{D}') = \mathit{Rank}(\mathcal{D}_1)$	By Lemma 2.16
$\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$	By Definition 2.15
$[e'/x]\,\mathcal{E}[x]$ satisfies (i)–(v)	By IH
$[e'/x]\,\mathcal{E}[x] = \mathcal{E}[e']$	By defn. of substitution (x new)
☞ $\mathcal{E}[e']$ satisfies (i)–(v)	By previous equation

- **Case** ΣE: Similar to the $\otimes$E case, using IH (iii) instead of IH (iv), and the substitution $i/a, e'/x$. □

2.6.3 Value inversion on $\rightarrow$, $*$, $\delta(i)$

For ordinary types (not property types) we have a value inversion lemma (also known as *genericity* or *canonical forms*) used in the respective elimination rule cases in the type safety proof (Thm. 2.21): for instance, that proof's case for $*E_1$ must invert $v : A_1 * A_2$ to obtain $v = (v_1, v_2)$ where $v_1 : A_1$. (The unit type **1** has no elimination rule, so it does not need an inversion lemma.)[10]

To get through the value inversion proof cases for the indefinite type elimination rules, such as $\vee$E, we use the same derivation measure μ that we just used to prove value definiteness. However, the statement of the proposition is simpler because, in those cases, we use value definiteness—not the IH—on the subterm e'. Thus we need not say the resulting derivation $\mathcal{D}'$ is smaller than $\mathcal{D}$; we do not even name the resulting derivation.

Lemma 2.18 (Value Inversion for $\rightarrow$, $*$, $\delta(i)$)**.** *If* $\mathcal{D} :: \cdot \vdash v : A$*, then*

(i) if $\cdot \vdash A \leq B_1 \rightarrow B_2$ *then* $v = \lambda x.\, e$ *where* $\cdot \vdash [v'/x]\, e : B_2$ *for any* $\vdash v' : B_1$*;*

(ii) if $\cdot \vdash A \leq B_1 * B_2$ *then* $v = (v_1, v_2)$ *where* $\cdot \vdash v_1 : B_1$ *and* $\cdot \vdash v_2 : B_2$*.*

(iii) if $\cdot \vdash A \leq \delta(i)$ *then* $v = c(v')$ *where* $\cdot \vdash \mathcal{S}(c) \uparrow B_1 \rightarrow \delta'(i')$ *and* $\cdot \vdash c(v') : \delta'(i')$ *and* $\cdot \vdash v' : B_1$ *and* $\cdot \vdash \delta'(i') \leq \delta(i)$*.*

Proof. By induction on the measure $\mu(\mathcal{D})$, where $\mathcal{D} :: \cdot \vdash v : A$. We show part (i), inversion on $\rightarrow$; for part (ii), inversion on $*$, all cases except $\rightarrow$I and $*$I are similar to the corresponding cases for part (i), while the $*$I case for (ii) is a slight simplification of the $\rightarrow$I case for (i). For part (iii), $\delta(i)$, all cases are similar to part (i) except $\rightarrow$I and δI; we show the δI case in full at the end.

Rules fixvar, fix, $\rightarrow$E, $*E_1$, $*E_2$, and δE cannot type values. For **1**I there is no way to derive $\cdot \vdash \mathbf{1} \leq B_1 \rightarrow B_2$, so the rule could not have been used; $*$I, δI, and $\top$I are similar.

Since the context is empty, the var case is impossible.

For sub, $\wedge E_{1,2}$, ΠE, $\supset$E, $\vee I_{1,2}$, ΣI, and $\otimes$I, we show that the type in each rule's premise is a subtype of A (immediate in sub, otherwise by the appropriate subtyping rule), then use transitivity of subtyping and apply the IH.

For $\perp$E, $\vee$E, ΣE, $\otimes$E and direct, in which the subject term is $\mathcal{E}[v']$, we use Theorem 2.17, yielding $\mathcal{D}' :: \cdot \vdash v' : C$ (for appropriate C) where $Rank(\mathcal{D}') = 0$. By an argument similar to that in the $\vee$E case of the proof of Theorem 2.17, we obtain a derivation $\mathcal{D}'_1 :: \cdot \vdash \mathcal{E}[v'] : A$ such that $Rank(\mathcal{D}'_1) < Rank(\mathcal{D})$, and therefore $\mu(\mathcal{D}'_1) < \mu(\mathcal{D})$, so we can apply the IH to $\mathcal{D}'_1$. We show the ΣE case below.

The contra case is excluded by Property 2.5.

The remaining cases are $\rightarrow$I, ΠI, $\wedge$I and $\supset$I.

- **Case** $\rightarrow$I: $\mathcal{D} :: \dfrac{x{:}A_1 \vdash e : A_2}{\cdot \vdash \lambda x.\, e : A_1 \rightarrow A_2}$

[10]The combination of the inversion properties for $\rightarrow$, $*$, and $d(i)$ into one lemma (Lemma 2.18) is not required; there is no interplay between these constructors in the type system, in particular, in the subtyping rules.

The result that $v = \lambda x.\, e$ is immediate. We have $\cdot \vdash A \leq B_1 \to B_2$ and $A = A_1 \to A_2$. By inversion, $\cdot \vdash B_1 \leq A_1$ and $\cdot \vdash A_2 \leq B_2$. For all v' such that $\cdot \vdash v' : B_1$:

$\cdot \vdash B_1 \leq A_1$	Above
$\cdot \vdash v' : B_1$	Assumption
$\cdot \vdash v' : A_1$	By sub
$\cdot \vdash v'/x : x{:}A_1$	By empty-σ and pvar-σ
$x{:}A_1 \vdash e : A_2$	Subd.
$\cdot \vdash [v'/x]e : [v'/x]A_2$	By Lemma 2.14
$\cdot \vdash [v'/x]e : A_2$	$x \notin FV(A_2)$
$\cdot \vdash A_2 \leq B_2$	Above
☞ $\cdot \vdash [v'/x]e : B_2$	By sub

- **Case** ΠI: $\mathcal{D} :: \dfrac{a{:}\gamma \vdash v : A'}{\cdot \vdash v : \Pi a{:}\gamma.\, A'}$

We have $A = \Pi a{:}\gamma.\, A'$.

$\cdot \vdash \Pi a{:}\gamma.\, A' \leq B_1 \to B_2$	Given
$\cdot \vdash [i/a]A' \leq B_1 \to B_2$ and $\cdot \vdash i : \gamma$	By inversion (only rule ΠL possible)
$\cdot \vdash i/a : a{:}\gamma$	By empty-σ and ivar-σ
$\mathcal{D}' :: a{:}\gamma \vdash v : A'$	Subd.
$\mathcal{D}'' :: \cdot \vdash [i/a]v : [i/a]A'$ $Rank(\mathcal{D}'') = Rank(\mathcal{D}')$ and $Size(\mathcal{D}'') \leq Size(\mathcal{D}')$	By Lemma 2.16
$\mu(\mathcal{D}'') \leq \mu(\mathcal{D}')$	By Definition 2.15
$\mathcal{D}'' :: \cdot \vdash v : [i/a]A'$	a not free in v
☞ $v = \lambda x.\, e$ and $\cdot \vdash [v'/x]\, e : B_2$ for any v' s.t. $\cdot \vdash v' : B_1$	By IH

- **Case** ⊃I: Similar to the ΠI case.

- **Case** ∧I: $\mathcal{D} :: \dfrac{\cdot \vdash v : A_1 \qquad \cdot \vdash v : A_2}{\cdot \vdash v : A_1 \wedge A_2}$

We have $A = A_1 \wedge A_2$, and $\cdot \vdash A \leq B_1 \to B_2$. The only subtyping rules that could have been used are $\wedge L_{1,2}$. These cases are symmetric; assume $\wedge L_2$. The premise of $\wedge L_2$ is $\cdot \vdash A_2 \leq B_1 \to B_2$. Applying the IH to $\cdot \vdash v : A_2$ yields $v = \lambda x.\, e$ where $\vdash [v'/x]\, e : B_2$ for any v' such that $\vdash v' : B_1$, which was to be shown.

- **Case** ΣE: $\mathcal{D} :: \dfrac{\cdot \vdash e' : \Sigma a{:}\gamma.\, C \qquad a{:}\gamma, x{:}C \vdash \mathcal{E}[x] : A}{\cdot \vdash \mathcal{E}[e'] : A}$

  By Theorem 2.17, there exists $\vdash i : \gamma$ such that $\cdot \vdash e' : [i/a]C$ by a rank-0 derivation. By Lemma 2.16, $\vdash [i/a, e'/x]\ \mathcal{E}[e'] : A$ by a derivation of the same rank as the subderivation $a{:}\gamma, x{:}C \vdash \mathcal{E}[x] : A$. That rank is one less than the rank of $\mathcal{D}$, so we can apply the IH, yielding the result.

That concludes parts (i) and (ii). For part (iii), we show the δI case:

- **Case** δI: $\mathcal{D} :: \dfrac{\cdot \vdash c : B_1 \to \delta'(i') \qquad \cdot \vdash v' : B_1}{\cdot \vdash c(v') : \delta'(i')}$

  The only rule that can conclude $\cdot \vdash c : B_1 \to \delta'(i')$ is $\mathcal{S}$-con, which has premise

☞	$\cdot \vdash \mathcal{S}(c) \uparrow B_1 {\to} \delta'(i')$	
	$A = \delta'(i')$	Given
☞	$\cdot \vdash c(v') : \delta'(i')$	Conclusion of derivation
☞	$v = c(v')$	Given
	$\cdot \vdash A \leq \delta(i)$	Given
☞	$\cdot \vdash v' : B_1$	Subderivation
☞	$\cdot \vdash \delta'(i') \leq \delta(i)$	By $A = \delta'(i')$

□

2.6.4 Lemmas for case

We prove two lemmas relating to constructors and matches:

- Lemma 2.19 (used in Lemma 2.20) allows inversion on judgments of the form $\cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$, obtaining $x{:}A_1 \vdash e : C$ where A_1 is appropriately related to A^{con};
- Lemma 2.20 shows that given a value $c(v) : \delta(i)$ and a typing of $ms = \ldots c(x) \Rightarrow e \ldots$, there is an A such that $v : A$ and $x{:}A \vdash e : C$, pursuant to showing that **case** $c(v)$ **of** $ms \mapsto_R [v/x]e$ preserves typing (in the δE case of Theorem 2.21).

Lemma 2.19. *If* $\cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$ *where* $\cdot \vdash A^{con} \uparrow A_1 \to \delta'(i')$ *and* $\delta' \preceq \delta$ *and* $\cdot \models i' \doteq i$ *then* $x{:}A_1 \vdash e : C$.

Proof. By induction on the size of $\mathcal{D} :: \cdot; c : A^{con}; c(x) : \delta(i) \vdash e : C$.

- **Case** δS-ct: $\mathcal{D} :: \dfrac{\delta'' \preceq \delta \qquad x{:}A', i'' \doteq i \vdash e : C}{\cdot; c : A' \to \delta''(i''); c(x) : \delta(i) \vdash e : C}$

  Only rule refl-↑ could have derived $\cdot \vdash A' {\to} \delta''(i'') \uparrow A_1 {\to} \delta'(i')$, so we have $A' = A_1$ and $\delta'' = \delta'$ and $i'' = i'$. The second subderivation is therefore $x{:}A_1, i' \doteq i \vdash e : C$. It is given that $\cdot \models i' \doteq i$. By Lemma 2.14 with substitution x/x we obtain $x{:}A_1 \vdash e : C$, which was to be shown.

- **Case** δF-ct: $\mathcal{D} :: \dfrac{\delta'' \not\preceq \delta \qquad x{:}A \vdash e\ \mathsf{ok}}{\cdot; c : A' \to \delta''(i); c(x) : \delta(i) \vdash e : C}$

  We have the premise $\delta'' \not\preceq \delta$. By analogy with the preceding case, $\delta'' = \delta'$, so $\delta' \not\preceq \delta$. This contradicts $\delta' \preceq \delta$, which was given: the case is impossible.

- **Case** $\wedge$-ct: $\mathcal{D} :: \dfrac{\cdot; c : A^{con}{}'_1; c(x) : \delta(i) \vdash e : C \qquad \cdot; c : A^{con}{}'_2; c(x) : \delta(i) \vdash e : C}{\cdot; c : A^{con}{}'_1 \wedge A^{con}{}'_2; c(x) : \delta(i) \vdash e : C}$

  Inversion on $\cdot \vdash A^{con}{}'_1 \wedge A^{con}{}'_2 \uparrow A_1 \to \delta'(i')$ yields $\cdot \vdash A^{con}{}'_k \uparrow A_1 \to \delta'(i')$ for some $k \in 1..2$. The result follows by IH on the kth premise.

- **Case** Π-ct, $\supset$-ct: Use inversion on $\cdot \vdash A^{con} \uparrow A_1 \to \delta'(i')$ to get $\cdot \vdash i : \gamma$ or $\cdot \models P$, and construct a substitution i/a or $\cdot$. Use Lemma 2.14 on a subderivation. Since $\overline{a{:}\gamma} = a{:}\gamma$ (and $\bar{\cdot} = \cdot$), the resulting derivation is no larger than the subderivation. The IH then gives the result. □

Lemma 2.20. *If* $\cdot \vdash c(v) : \delta(i)$ *and* $\cdot \vdash ms :_{\delta(i)} C$ *and* $ms = \ldots c(x) \Rightarrow e \ldots$ *then there exists* A *such that* $x{:}A \vdash e : C$ *where* $\cdot \vdash v : A$.

Proof. By induction on the derivation of $\cdot \vdash ms :_{\delta(i)} C$. Two rules can derive such judgments:

- **Case** emptyms: $ms = \cdot$, but it is given that ms includes $c(x) \Rightarrow e$. This rule could not have been used.

- **Case** casearm: $\mathcal{D} :: \dfrac{\cdot; c' : \mathcal{S}(c'); c'(x') : \delta(i) \vdash e' : C \qquad \cdot \vdash ms' :_{\delta(i)} C}{\cdot \vdash (c'(x') \Rightarrow e') \mid ms' \ :_{\delta(i)}\ C}$

  If $c' \neq c$: It is given that $ms = (c'(x') \Rightarrow e') \mid ms'$ includes $c(x) \Rightarrow e$. Since $c' \neq c$, ms' must include $c(x) \Rightarrow e$. The result follows by IH on $\cdot \vdash ms' :_{\delta(i)} C$.

  If $c' = c$: By the assumption that each constructor appears in exactly one case arm, $c'(x') \Rightarrow e' = c(x) \Rightarrow e$. Therefore the first subderivation is

  $$\cdot; c : \mathcal{S}(c); c(x) : \delta(i) \vdash e : C.$$

  $\cdot \vdash \delta(i) \leq \delta(i)$ by Lemma 2.9. By Property 2.5, $\cdot \not\models \bot$. Thus we can use Lemma 2.18 on $\cdot \vdash c(v) : \delta(i)$ to yield A and $\delta'(i')$ such that $\cdot \vdash \mathcal{S}(c) \uparrow A{\to}\delta'(i')$ where $\cdot \vdash v : A$ and $\cdot \vdash \delta'(i') \leq \delta(i)$. By inversion on the latter, $\delta' \preceq \delta$ and $\cdot \models i' \doteq i$.

  We can now apply Lemma 2.19 to show $x{:}A \vdash e : C$, which with $\cdot \vdash v : A$ constitutes the result. □

2.7 Type preservation and progress

Having proved value definiteness, value inversion, and properties of case expressions, we are ready to prove type safety. We prove the preservation and progress theorems simultaneously; we could

prove them separately, but the proofs would share so much structure as to be more cumbersome than the simultaneous proof.[11] Note that while we have elided any explicit effects from the present system, the analysis in [DP00] applies in this setting. Again we use the derivation measure $\mu(\mathcal{D})$, which allows application of the IH after substitution in a subderivation in the direct, $\bot$E, $\vee$E, ΣE and $\otimes$E cases.

Theorem 2.21 (Type Preservation and Progress). *If* $\mathcal{D} :: \cdot \vdash e : C$ *then either*

(1) e value*, or*

(2) there exists e' *such that* $e \mapsto e'$ *and* $\vdash e' : C$.

Proof. By induction on the measure $\mu(\mathcal{D})$ of the derivation $\mathcal{D} :: \cdot \vdash e : C$. If e value, the result is immediate. So suppose e is not a value.

Rules **1**I, $\rightarrow$I, $\wedge$I, $\top$I, ΠI, and $\supset$I can type only values, and are thus excluded. The context is empty, so var and fixvar are excluded, and by Property 2.5 rule contra cannot have been used.

The case for fix uses Lemma 2.14. For sub, $\vee I_{1,2}$, ΣI, $\otimes$I, $\wedge E_{1,2}$, $\supset$E and δI we simply use the IH and apply the rule again.

- **Case** $\rightarrow$E: $\mathcal{D} :: \dfrac{\cdot \vdash e_1 : A \rightarrow C \qquad \cdot \vdash e_2 : A}{\cdot \vdash e_1 e_2 : C}$

  If e_1 is not a value, use the IH with $\cdot \vdash e_1 : A \rightarrow C$, yielding e_1' such that $e_1 \mapsto e_1'$ and $\cdot \vdash e_1' : A \rightarrow C$. Given $e_1 \mapsto e_1'$ we have $e_1\, e_2 \mapsto e_1' e_2$. We have a subderivation of $\cdot \vdash e_2 : A$. By $\rightarrow$E, $\cdot \vdash e_1' e_2 : C$.

  The situation is symmetric if e_1 is a value and e_2 is not.

  If both e_1 and e_2 are values: we have $\cdot \vdash e_1 : A \rightarrow C$ and $\cdot \vdash e_2 : A$. By Lemma 2.18, $e_1 = \lambda x.\, e_0$ where, for all v' such that $\cdot \vdash v' : A$, it is the case that $\cdot \vdash [v'/x]\, e_0 : C$. Let $v' = e_2$. Then we have $\cdot \vdash [e_2/x]\, e_0 : C$. Finally, $e_1 = \lambda x.\, e_0$ and e_2 is a value, so $e_1 e_2 \mapsto [e_2/x]\, e_0$.

- **Case** $*$I: $\mathcal{D} :: \dfrac{\cdot \vdash e_1 : C_1 \qquad \cdot \vdash e_2 : C_2}{\cdot \vdash (e_1, e_2) : C_1 * C_2}$

  At least one of e_1, e_2 is not a value. Suppose e_1 is not.

	$e_1 \mapsto e_1'$ and $\cdot \vdash e_1' : C_1$	By IH
☞	$(e_1, e_2) \mapsto (e_1', e_2)$	By ev-context
	$\cdot \vdash e_2 : C_2$	Subd.
☞	$\cdot \vdash (e_1', e_2) : C_1 * C_2$	By $*$I

  The case where e_1 is a value and e_2 is not is similar.

[11] Our semantics is deterministic, so the combined form is useful. In a nondeterministic semantics, the guarantee of the combined form would be too weak: saying that *there exists* a well-typed term to which the present term steps does not preclude stepping to some other ill-typed term.

- **Case** δE: $\mathcal{D} :: \dfrac{\cdot \vdash e_0 : \delta(i) \qquad \cdot \vdash ms :_{\delta(i)} C}{\cdot \vdash \textbf{case } e_0 \textbf{ of } ms : C}$

  If e_0 is not a value the case is straightforward. If e_0 is a value: We have subderivations

  $$\cdot \vdash e_0 : \delta(i) \quad \text{and} \quad \cdot \vdash ms :_{\delta(i)} C$$

  By Lemma 2.18, $e_0 = c(v)$ for some v. We assume there is exactly one case arm for each constructor, so $ms = \ldots c(x) \Rightarrow e_c \ldots$ for some x, e_c. By Lemma 2.20, $x{:}A \vdash e_c : C$ where $\cdot \vdash v : A$.

  We have **case** e_0 **of** $ms =$ **case** $c(v)$ **of** $\ldots c(x) \Rightarrow e_c \ldots$. By the reduction rule in Figure 2.4, **case** $c(v)$ **of** $\ldots c(x) \Rightarrow e_c \ldots \mapsto_R [v/x]e_c$. By Lemma 2.14, $\cdot \vdash [v/x]\, e_c : C$. Let $e' = [v/x]\, e_c$ and we have the result.

- **Case** $*E_1$: $\mathcal{D} :: \dfrac{\cdot \vdash e_0 : C * C_2}{\cdot \vdash \textbf{fst}(e_0) : C}$

  If e_0 is not a value, the case is straightforward. If e_0 is a value:

	$\cdot \vdash e_0 : C * C_2$	Subderivation
	$\cdot \vdash C * C_2 \leq C * C_2$	By Lemma 2.9
	$e_0 = (v_1, v_2)$	By Lemma 2.18
☞	$\cdot \vdash v_1 : C$	(By Lemma 2.18)
☞	$\textbf{fst}((v_1, v_2)) \mapsto v_1$	

  The $*E_2$ case is similar.

- **Case** $\supset$E: $\mathcal{D} :: \dfrac{\cdot \vdash e : P \supset C \qquad \cdot \models P}{\cdot \vdash e : C}$

  By IH, $e \mapsto e'$ and $\cdot \vdash e' : P \supset C$. Applying $\supset$E yields $\cdot \vdash e' : C$, which was to be shown.

- **Case** $\otimes$I: $\mathcal{D} :: \dfrac{\cdot \vdash e : C' \qquad \cdot \models P}{\cdot \vdash e : P \otimes C'}$

  Similar to the previous case.

For direct, $\bot$E, $\vee$E, ΣE and $\otimes$E, which type an evaluation context $\mathcal{E}[e']$, we proceed thus:

- If the whole term $\mathcal{E}[e']$ is a value, we have the result.
- If e' is not a value:

  (1) Apply the IH to $\cdot \vdash e' : D$ to obtain $e' \mapsto e''$ with $\vdash e'' : D$.

  (2) From $e' \mapsto e''$, use ev-context to show $\mathcal{E}[e'] \mapsto \mathcal{E}[e'']$.

  (3) Reapply the rule, with premise $\vdash e'' : D$, to yield $\vdash \mathcal{E}[e''] : C$.

- If e' is a value (but $\mathcal{E}[e']$ is not), use value definiteness (Theorem 2.17), yielding a contradiction (the $\bot$E case), or a new derivation (the direct, $\vee$E, ΣE, and $\otimes$E cases) which can be substituted in another premise.

The last subcase, where e' is a value and $\mathcal{E}[e']$ is not, is the most interesting; we show it for $\bot$E, $\vee$E, ΣE and $\otimes$E. The direct case is similar; it uses part (v) of Theorem 2.17.

- **Case** $\bot$E: $\mathcal{D} :: \dfrac{\cdot \vdash e' : \bot \qquad \cdot \vdash \mathcal{E}[e']\ \mathsf{ok}}{\cdot \vdash \mathcal{E}[e'] : C}$

  We have $\cdot \vdash e' : \bot$ as a subderivation. By Lemma 2.9, $\cdot \vdash \bot \leq \bot$, but by Theorem 2.17 (i), $\cdot \nvdash \bot \leq \bot$, a contradiction. Therefore $\bot$E could not have derived $\mathcal{D}$.

- **Case** $\vee$E: $\mathcal{D} :: \dfrac{\overset{\mathcal{D}_0}{\cdot \vdash e' : A \vee B} \qquad \overset{\mathcal{D}_1}{x{:}A \vdash \mathcal{E}[x] : C} \qquad \overset{\mathcal{D}_2}{y{:}B \vdash \mathcal{E}[y] : C}}{\cdot \vdash \mathcal{E}[e'] : C}$

  e' value is given. We have $\mathcal{D}_0 :: \cdot \vdash e' : A \vee B$ as a subderivation. By Theorem 2.17 (ii), either $\cdot \vdash e' : A$ or $\cdot \vdash e' : B$ by a derivation $\mathcal{D}'$ of rank 0. Assume $\vdash e' : A$ (the other case is symmetric). $\mathcal{D}_1 :: x{:}A \vdash \mathcal{E}[x] : C$ is a subderivation of rank $\mathit{Rank}(\mathcal{D}_1) < \mathit{Rank}(\mathcal{D})$. By empty-$\sigma$ and pvar-σ, $\cdot \vdash e'/x : x{:}A$. By Lemma 2.16, $\mathcal{D}_1' :: \cdot \vdash \mathcal{E}[e'] : C$ where $\mathit{Rank}(\mathcal{D}_1') = \mathit{Rank}(\mathcal{D}_1)$, which is less than $\mathit{Rank}(\mathcal{D})$, so we can apply the IH to $\mathcal{D}_1'$, obtaining $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$.

- **Case** ΣE: $\mathcal{D} :: \dfrac{\overset{\mathcal{D}_0}{\cdot \vdash e' : \Sigma a{:}\gamma.\, A} \qquad \overset{\mathcal{D}_1}{a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C}}{\cdot \vdash \mathcal{E}[e'] : C}$

e' value	Given
$\mathcal{D}_0 :: \cdot \vdash e' : \Sigma a{:}\gamma.\, A$	Subderivation
$\exists i.\ \ \mathcal{D}_0' :: \cdot \vdash e' : [i/a]A$ where $\cdot \vdash i : \gamma$ and $\mathit{Rank}(\mathcal{D}_0') = 0$	} By Theorem 2.17 (iii)
$\cdot \vdash \cdot : \cdot$	By empty-σ
$\cdot \vdash i/a : a{:}\gamma$	By ivar-σ
$\cdot \vdash i/a, e'/x : a{:}\gamma, x{:}A$	By pvar-σ
$\mathit{Rank}(i/a, e'/x) = 0$	$\mathit{Rank}(\mathcal{D}_0') = 0$
$\mathcal{D}_1 :: a{:}\gamma, x{:}A \vdash \mathcal{E}[x] : C$	Subderivation
$\mathcal{D}_1' :: \cdot \vdash [i/a, e'/x]\, \mathcal{E}[x] : [i/a, e'/x]C$; $\mathit{Rank}(\mathcal{D}_1') = \mathit{Rank}(\mathcal{D}_1)$	} By Lemma 2.16
$\mathcal{D}_1' :: \cdot \vdash [e'/x]\, \mathcal{E}[x] : C$	i is a new index variable
$\mathcal{D}_1' :: \cdot \vdash \mathcal{E}[e'] : C$	x is new, so not free in $\mathcal{E}$

$Rank(\mathcal{D}_1) < Rank(\mathcal{D})$	By Definition 2.15
$Rank(\mathcal{D}_1') < Rank(\mathcal{D})$	$Rank(\mathcal{D}_1') = Rank(\mathcal{D}_1)$
$\mu(\mathcal{D}_1') < \mu(\mathcal{D})$	By Definition 2.15

☞ $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$ By IH

- **Case** $\otimes$E: $\mathcal{D} :: \dfrac{\cdot \vdash e' : P \otimes A \qquad P, x{:}A \vdash \mathcal{E}[x] : C}{\cdot \vdash \mathcal{E}[e'] : C}$

e' value	Given
$\mathcal{D}_0 :: \cdot \vdash e' : P \otimes A$	Subderivation
$\mathcal{D}_0' :: \cdot \vdash e' : A$ where $\cdot \models P$ and $Rank(\mathcal{D}_0') = 0$	By Theorem 2.17 (iv)
$\cdot \vdash \cdot : \cdot$	By empty-σ
$\cdot \vdash \cdot : P$	By prop-σ
$\cdot \vdash e'/x : P, x{:}A$	By pvar-σ
$Rank(e'/x) = 0$	$Rank(\mathcal{D}_0') = 0$
$\mathcal{D}_1 :: P, x{:}A \vdash \mathcal{E}[x] : C$	Subderivation
$\mathcal{D}_1' :: \cdot \vdash [e'/x]\,\mathcal{E}[x] : [e'/x]C$ and $Rank(\mathcal{D}_1') = Rank(\mathcal{D}_1)$	By Lemma 2.16
$\mathcal{D}_1' :: \cdot \vdash \mathcal{E}[e'] : C$	x is new, so not free in $\mathcal{E}$
$Rank(\mathcal{D}_1) < Rank(\mathcal{D})$	By Definition 2.15
$Rank(\mathcal{D}_1') < Rank(\mathcal{D})$	$Rank(\mathcal{D}_1') = Rank(\mathcal{D}_1)$
$\mu(\mathcal{D}_1') < \mu(\mathcal{D})$	By Definition 2.15

☞ $\mathcal{E}[e'] \mapsto e''$ and $\cdot \vdash e'' : C$ By IH ☐

2.8 Related work

Refinement using indexed datatypes and dependent function types with indices drawn from a decidable constraint domain was proposed by Xi and Pfenning [XP99]. Their language did not introduce pure property types, requiring syntactic markers to eliminate existentials. Since this was unrealistic for many programs, Xi [Xi98] presented an algorithm allowing existential elimination at every binding site after translation to a let-normal form. Some of our results can be seen as a *post hoc* justification for that basic idea; see the remarks at the end of Section 2.4.5—and Chapter 5.

Pierce [Pie91b] gave examples of programming with intersection and union types in a pure λ-calculus using a typechecking mechanism that relied on syntactic markers. MacQueen et al. [MPS86] appear to be the first to publish work on union types, in their development of a model of recursive polymorphic types. The first systematic study of unions in a type assignment framework by

Barbanera et al. [BDCd95] identified a number of problems, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted (as it was in MacQueen et al.). That study also provided a framework for our more specialized study of a call-by-value language with possible effects.

Van Bakel et al. [vBDCdM00] showed that the minimal relevant logic B+ yields a type assignment system for the pure call-by-value λ-calculus; conjunction and disjunction become intersection and union, respectively. In their $\vee$-elimination rule, the subexpression may appear multiple times but must be a value; this rule is sound but impractical (see Section 2.4.1).

Litvinov [Lit03] used pure intersections and unions in typechecking programs in the object-oriented language Cecil. Igarashi and Nagira [IN06] extended Java with union types. Their type system includes both an explicit case statement (based on objects' runtime tags) and "direct access" to object members: given two classes C1 and C2, neither of which is a subclass of the other, and an `x` declared as type C1 $\vee$ C2, the access `x.m` is permitted as long as both C1 and C2 have a member `m`. These are not exactly tagged unions in the usual sense, nor are they exactly pure unions like those in our system—they are a form of pure union over the class hierarchy, which is *itself* a tagged union.

Finally, forms of union types have been used for control flow analysis and soft typing, as mentioned in Section 1.6.1.

2.9 Conclusion

We have designed a system of property types for the purpose of checking program invariants in call-by-value languages. We have presented the system as it was designed: incrementally, with each type constructor added orthogonally to an intermediate system that is itself sound and logically motivated. For both the definite and indefinite types, we have formulated rules that are not only sound but internally regular: the differences among $\vee$E, $\bot$E, ΣE, direct are logical consequences of the type constructor's arity. The remarkable feature shared by all four rules is that typing may proceed in evaluation order, constituting a less *ad hoc* alternative to Xi's conversion to let-normal form. Lastly, we have formulated properties of *definiteness* of judgments, substitutions, and values, vital for our proof of type safety.

The pure type assignment system presented here is undecidable. The present system can be used to verify progress and preservation after erasure of all type annotations, and is the basis of soundness in the bidirectional system in Chapter 3. In particular, it verifies that typechecked programs do not need to carry types at runtime.

While we have elided any explicit effects from the present system for the sake of brevity, the analysis in [DP00] applies to this setting and the present system.

Chapter 3

A tridirectional type system[1]

3.1 Introduction

Chapter 2 developed a system of pure type assignment designed for call-by-value languages with effects and proved progress and type preservation. As a pure type assignment system, where terms do not contain any types at all, it is inherently undecidable due to unrestricted intersection types [CDCV81, AC98].

In this chapter we develop an annotation discipline and typechecking algorithm for the type assignment system. The major contribution is the type system itself, which contains several novel ideas, including an extension of the paradigm of bidirectional typechecking to union and existential types, leading to the *tridirectional system*. While type soundness follows immediately by erasure of annotations, completeness requires that we insert *contextual typing annotations* reminiscent of principal typings [Jim95, Wel02]. Decidability is not obvious; we prove it by showing that a slightly altered *left tridirectional system* is decidable (and sound and complete with respect to the tridirectional system).

The basic underlying idea is *bidirectional checking* [PT98] of programs containing some type annotations, combining *type synthesis* with *type analysis*, first adapted to property types by Davies and Pfenning [DP00]. Synthesis generates a type for a term from its immediate subterms. Logically, this is appropriate for destructors (or *elimination forms*) of a type. For example, the first product elimination passes from $e : A * B$ to $\mathbf{fst}(e) : A$. Therefore, if we can generate $A * B$ we can extract A. Dually, analysis verifies that a term has a given type by verifying appropriate types for its immediate subterms. Logically, this is appropriate for constructors (or *introduction forms*) of a type. For example, to verify that $(\lambda x.\, e) : A \to B$ we assume $x : A$ and then verify $e : B$. Bidirectional checking works for both the native types of the underlying programming language and the layer of property types we construct over it.

However, the simple bidirectional model is not sufficient for the indefinite property types: unions, existential quantification, the empty type and asserting types. This is because the program lacks the requisite structure. For example, if we synthesize the union $A \vee B$ for an expression e, we now need to distinguish the cases: the value of e might have type A or it might have type B.

[1]This chapter is based on joint work with Frank Pfenning [DP04a].

Determining the proper scope of this case distinction depends on how e is used, that is, the position in which e occurs. This means we need a "third direction" (whence the name *tridirectional*): we might need to move to a subexpression e, synthesize its type, and only then analyze the expression $\mathcal{E}[e]$ surrounding it.

Since the tridirectional type system (like the bidirectional one) requires annotations, we want to know that any program well typed in the type assignment system can be annotated so that it is also well typed in the tridirectional system. But with intersection types, such a completeness property does not hold for the usual form of type annotation $(e : A)$ (as previously noted [Pie91a, Dav05a, WH02]), a problem exacerbated by scoping issues arising from quantified types. We therefore extend the notion of type annotation to *contextual typing annotation*, $(e : \Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n)$, in which the programmer can write several context/type pairs. The idea is that an annotation $\Gamma_k \vdash A_k$ may be used when e is checked in a context matching Γ_k. This idea might also be applicable to arbitrary rank polymorphism, a possibility we leave to future work.

Unlike the bidirectional system, the indefinite property types that necessitate the third direction make decidability of typechecking nontrivial. Two ideas come to the rescue. First, to preserve type safety in a call-by-value language with effects, the type of a subterm e can only be analyzed if the term containing it has the form $\mathcal{E}[e]$ for some evaluation context $\mathcal{E}$, reducing[2] the nondeterminism; this was a key observation in Chapter 2. Second, one never needs to visit a subterm more than once in the same derivation: the system which enforces this is sound and complete.

Section 3.2 presents a simple bidirectional type system. Section 3.3 adds refinements and a rich set of types including intersections and unions, using tridirectional rules; this is the *simple tridirectional system*. In Section 3.4, we explain our form of typing annotation and prove that the simple tridirectional system is complete with respect to the type assignment system. Section 3.5 restricts the tridirectional rules and compensates by introducing *left rules* to yield a *left tridirectional system*, reducing nondeterminism. We prove soundness and completeness with respect to the simple tridirectional system, prove decidability, and use the results in Chapter 2 to prove type safety. Finally, we discuss related work (Section 3.6) and conclude (Section 3.7).

3.2 The core language

In a pure type assignment system, the typing judgment is $e : A$, where e contains no types (eliding contexts for the moment). In a bidirectional type system, we have two typing judgments: $e \uparrow A$, read e *synthesizes* A, and $e \downarrow A$, read e *checks against* A. An implementation of such a system consists of two mutually recursive functions: the first, corresponding to $e \uparrow A$, takes the term e and either returns A or fails; the second, corresponding to $e \downarrow A$, takes the term e *and* a type A and succeeds (returning nothing) or fails. This raises a question: Where do the types in the judgments $e \downarrow A$ come from? More generally: what are the design principles behind a bidirectional type system?

Avoiding unification or similar techniques associated with full type inference is fundamental to the design of our bidirectional system. The motivation for this is twofold. First, for highly expressive systems such as those considered here, full type inference is often undecidable, so we

[2] However, the choice of $\mathcal{E}$ is still quite nondeterministic, an important issue we address in Chapter 5.

need less automatic and more robust methods. Second, since unification globally propagates type information, it is often difficult to pinpoint the source of type errors.

We think of the process of bidirectional typechecking as a bottom-up construction of a typing derivation, either of $e \uparrow A$ or $e \downarrow A$. Given that we want to avoid unification and similar techniques, we need each inference rule to be *mode correct*, terminology borrowed from logic programming [RP96]. That is, for any rule with conclusion $e \uparrow A$ we must be able to determine A from the information in the premises. Conversely, if we have a rule with premise $e \downarrow A$, we must be able to determine A before traversing e.

However, mode correctness by itself is only a consistency requirement, not a design principle. We find such a principle in the realm of logic, and transfer it to our setting. In natural deduction, we distinguish *introduction rules* and *elimination rules*. An introduction rule specifies how to infer a proposition from its components; when read bottom-up, it decomposes the proposition. For example, the introduction rule for the conjunction $A * B$ decomposes it to the goals of proving A and B. Therefore, a rule that checks a term *against* $A * B$ using an introduction rule will be mode correct.

$$\frac{\Gamma \vdash e_1 \downarrow A_1 \qquad \Gamma \vdash e_2 \downarrow A_2}{\Gamma \vdash (e_1, e_2) \downarrow A_1 * A_2} *\mathrm{I}$$

Conversely, an elimination rule specifies how to use the fact that a certain proposition holds; when read top-down, it decomposes a proposition. For example, the two elimination rules for the conjunction $A * B$ decompose it to A and B, respectively. Therefore, a rule that infers a type for a term using an elimination rule will be mode correct.

$$\frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{fst}(e) \uparrow A} *\mathrm{E}_1 \qquad \frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{snd}(e) \uparrow B} *\mathrm{E}_2$$

If we employ this design principle throughout, the constructors (corresponding to the introduction rules) for the elements of a type are *checked against* a given type, while the destructors (corresponding to the elimination rules) for the elements of a type *synthesize* their type. This leads to the following rules for functions, in which rule $\rightarrow$I checks against $A \rightarrow B$ and rule $\rightarrow$E synthesizes the type $A \rightarrow B$ of its subject

$$\frac{\Gamma, x{:}A \vdash e \downarrow B}{\Gamma \vdash \lambda x.\, e \downarrow A \rightarrow B} \rightarrow\mathrm{I} \qquad \frac{\Gamma \vdash e_1 \uparrow A \rightarrow B \qquad \Gamma \vdash e_2 \downarrow A}{\Gamma \vdash e_1(e_2) \uparrow B} \rightarrow\mathrm{E}$$

What do we do when the different judgment directions meet? If we are trying to check $e \downarrow A$ then it suffices to synthesize a type $e \uparrow A'$ and check that $A' = A$. More generally, in a system with subtyping, it suffices to know that every value of type A' also has type A, that is, $A' \leq A$.

$$\frac{\Gamma \vdash e \uparrow A' \qquad \Gamma \vdash A' \leq A}{\Gamma \vdash e \downarrow A} \mathsf{sub}$$

In the opposite direction, if we want to synthesize a type for e but can only check e against a given type, then we do not have enough information. In the realm of logic, such a step would correspond to a proof that is not in normal form (and might not have the subformula property [Pra65, p. 53]. The straightforward solution would be to allow source expressions $(e : A)$ via a rule

$$\frac{\Gamma \vdash e \downarrow A}{\Gamma \vdash (e : A) \uparrow A}$$

Unfortunately, this is not general enough due to the presence of intersections and universally and existentially quantified property types. We discuss the issues and our solution in detail in Section 3.4. For now, only normal terms will typecheck in our system. These correspond exactly to normal proofs in natural deduction [Pra65]. We can therefore already pinpoint where annotations will be required in the full system: exactly where the term is not normal. This will be the case where destructors are applied to constructors (that is, as redexes) and at certain **let** forms.

In addition we permit datatypes δ with constructors $c(e)$ and corresponding case expressions **case** e **of** ms, where the match expressions ms have the form $c_1(x_1) \Rightarrow e_1 \mid \ldots c_n(x_n) \Rightarrow e_n$. The constants c are the constructors and **case** the destructor of elements of type δ. This means expressions $c(e)$ are checked against a type, while the subject of a **case** must synthesize its type. Assuming constructors have type $A \to \delta$, this yields the following rules.

$$\frac{c : A \to \delta \qquad \Gamma \vdash e \downarrow A}{\Gamma \vdash c(e) \downarrow \delta}\,\delta\mathrm{I} \qquad \frac{\Gamma \vdash e \uparrow \delta \qquad \Gamma \vdash ms \downarrow_\delta B}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms \downarrow B}\,\delta\mathrm{E}$$

$$\frac{}{\Gamma \vdash \cdot \downarrow_\delta C} \qquad \frac{c : A \to \delta \qquad \Gamma, x{:}A \vdash e \downarrow B \qquad \Gamma \vdash ms \downarrow_\delta B}{\Gamma \vdash c(x) \Rightarrow e \mid ms\ \downarrow_\delta B}$$

As in Chapter 2, the left-hand sides of a **case** expression with subject δ must cover all constructors for a type δ (we will remove this requirement when we extend the language of patterns in Chapter 4). Note that in the elimination rule δE, we move from $e \uparrow \delta$ to $x{:}A$ (which should be read $x{\uparrow}A$) , checking each branch against C.

In addition we have fixed points, which involve both directions: to check **fix** $u.\, e \downarrow A$ we assume $u{:}A$ (which should be read $u{\uparrow}A$) then the check e against A. Here we have a new form of variable u that does not stand for a value, but for an arbitrary term, because the reduction form for fixed point expressions reduces **fix** $u.\, e$ to [**fix** $u.\, e/u$] e (the substitution of **fix** $u.\, e$ for u in e). We do not exploit this generality here, but our design is clearly consistent with common syntactic restriction on the formation of fixed points in call-by-value languages; in particular, the SML declaration **fun** $f\ x\ =\ e$ corresponds to **fix** $u.\, \lambda x.\, e$.

The syntax and semantics of our core language is given in Figure 3.1. Evaluation contexts $\mathcal{E}$ and the small-step call-by-value semantics are unchanged from Chapter 2.

Figure 3.2 shows the subtyping and typing rules for the initial language. The subtyping rules are standard except for the presence of the context Γ, used by the subtyping rules for index refinements and index quantifiers, which we add in the next section. Variables must appear in Γ, so var is a synthesis rule deriving $x \uparrow A$. The subsumption rule sub is an analysis rule deriving $e \downarrow B$, but its first premise is a synthesis rule $e \uparrow A$. This means both A and B are available when the subtyping judgment $A \leq B$ is invoked; no complex constraint management is necessary. For introduction and elimination rules, we follow the principles outlined above. Note that in practice, in applications $e_1 e_2$, the function e_1 will usually be a variable or, in a curried style, another application—since we synthesize types for these, $e_1 e_2$ itself needs no annotation.

Ours is not the only plausible formulation of bidirectionality. Xi [Xi98] used a contrasting style, in which several introduction forms have synthesis rules as well as checking rules, for example:

$$
\begin{array}{rrcl}
\text{Types} & A, B, C, D & ::= & \mathbf{1} \mid A \to B \mid A * B \mid \delta \\
\text{Terms} & e & ::= & x \mid u \mid \lambda x.\, e \mid e_1\, e_2 \mid \mathbf{fix}\ u.\, e \\
& & & \mid () \mid (e_1, e_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \\
& & & \mid c(e) \mid \mathbf{case}\ e\ \mathbf{of}\ ms \\
\text{Matches} & ms & ::= & \cdot \mid c(x) \Rightarrow e \mid ms \\
\text{Values} & v & ::= & x \mid \lambda x.\, e \mid () \mid (v_1, v_2) \\
\text{Eval. contexts} & \mathcal{E} & ::= & [\,] \mid \mathcal{E}(e) \mid v(\mathcal{E}) \\
& & & \mid (\mathcal{E}, e) \mid (v, \mathcal{E}) \mid \mathbf{fst}(\mathcal{E}) \mid \mathbf{snd}(\mathcal{E}) \\
& & & \mid c(\mathcal{E}) \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ ms
\end{array}
$$

$$
\frac{e' \mapsto_R e''}{\mathcal{E}[e'] \mapsto \mathcal{E}[e'']}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
(\lambda x.\, e)\, v & \mapsto_R & [v/x]\, e & \mathbf{fst}(v_1, v_2) & \mapsto_R & v_1 \\
\mathbf{fix}\ u.\, e & \mapsto_R & [\mathbf{fix}\ u.\, e \,/\, u]\, e & \mathbf{snd}(v_1, v_2) & \mapsto_R & v_2
\end{array}
$$

$$
\mathbf{case}\ c(v)\ \mathbf{of}\ \ldots c(x) \Rightarrow e \ldots \mapsto_R [v/x]\, e
$$

Figure 3.1: Syntax and semantics of the core language

$$
\frac{\Gamma \vdash B_1 \le A_1 \qquad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 \to A_2 \le B_1 \to B_2} \to
\qquad
\frac{}{\Gamma \vdash \mathbf{1} \le \mathbf{1}} \mathbf{1}
\qquad
\frac{\Gamma \vdash A_1 \le B_1 \qquad \Gamma \vdash A_2 \le B_2}{\Gamma \vdash A_1 * A_2 \le B_1 * B_2} *
\qquad
\frac{}{\Gamma \vdash \delta \le \delta} \delta
$$

$$
\frac{\Gamma(x) = A}{\Gamma \vdash x \uparrow A} \mathsf{var}
\qquad
\frac{\Gamma, x{:}A \vdash e \downarrow B}{\Gamma \vdash \lambda x.\, e \downarrow A \to B} \to\!\mathrm{I}
\qquad
\frac{\Gamma \vdash e_1 \uparrow A \to B \qquad \Gamma \vdash e_2 \downarrow A}{\Gamma \vdash e_1 e_2 \uparrow B} \to\!\mathrm{E}
$$

$$
\frac{\Gamma \vdash e \uparrow A \qquad \Gamma \vdash A \le B}{\Gamma \vdash e \downarrow B} \mathsf{sub}
\qquad
\frac{\Gamma(u) = A}{\Gamma \vdash u \uparrow A} \mathsf{fixvar}
\qquad
\frac{\Gamma, u{:}A \vdash e \downarrow A}{\Gamma \vdash \mathbf{fix}\ u.\, e \downarrow A} \mathsf{fix}
$$

$$
\frac{\Gamma \vdash e_1 \downarrow A_1 \qquad \Gamma \vdash e_2 \downarrow A_2}{\Gamma \vdash (e_1, e_2) \downarrow A_1 * A_2} *\mathrm{I}
\qquad
\frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{fst}(e) \uparrow A} *\mathrm{E}_1
\qquad
\frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{snd}(e) \uparrow B} *\mathrm{E}_2
\qquad
\frac{}{\Gamma \vdash () \downarrow \mathbf{1}} \mathbf{1}\mathrm{I}
$$

$$
\frac{\overline{\Gamma} \vdash c : A \to \delta \qquad \Gamma \vdash e \downarrow A}{\Gamma \vdash c(e) \downarrow \delta} \delta\mathrm{I}
\qquad
\frac{\Gamma \vdash e \uparrow \delta \qquad \Gamma \vdash ms \downarrow_\delta B}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms \downarrow B} \delta\mathrm{E}
$$

$$
\frac{}{\Gamma \vdash \cdot \downarrow_\delta B}
\qquad
\frac{c : A \to \delta \qquad \Gamma, x{:}A \vdash e \downarrow C \qquad \Gamma \vdash ms \downarrow_\delta C}{\Gamma \vdash c(x) \Rightarrow e \mid ms \downarrow_\delta C}
$$

Figure 3.2: Subtyping and typing in the core language

$$\frac{\Gamma \vdash e_1 \uparrow A_1 \qquad \Gamma \vdash e_2 \uparrow A_2}{\Gamma \vdash (e_1, e_2) \uparrow A_1 * A_2}$$

Xi's formulation does reduce the number of annotations; for example, in **case** (x, y) **of** ... the pair (x, y) must synthesize, but under our formulation (x, y) never synthesizes and so requires an annotation. However, ours seems to be the *simplest* plausible formulation and has a logical foundation in the notion of introduction and elimination forms corresponding to constructors and destructors for elements of a type under the Curry-Howard isomorphism. Consequently, a systematic extension should suffice to add further language constructs. Furthermore, any term in normal form will need no annotation except at the outermost level, so we should need annotations in few places besides function definitions. In any case, if a system based on our formulation turns out to be inconvenient, adding rules such as the one above should not be difficult.

3.3 Property types

The types present in the language so far are tied to constructors and destructors of terms. For example, the type $A \to B$ is realized by constructor $\lambda x.\, e$ and destructor $e_1 e_2$, related to the introduction and elimination forms of $\to$ by a Curry-Howard correspondence.

In this section we are concerned with expressing richer properties of terms already present in the language. The only change to the term language is to add typing annotations, discussed in Section 3.4; otherwise, only the language of types is enriched:

$$\begin{array}{lll} \text{Types} & A, B, C ::= & \ldots \\ & & \mid \delta(i) \\ & & \mid A \wedge B \mid \top \mid \Pi a{:}\gamma.\, A \mid P \supset A \\ & & \mid A \vee B \mid \bot \mid \Sigma a{:}\gamma.\, A \mid P \otimes A \end{array}$$

The basic properties are data structure invariants, that is, properties of terms of the form $c(e)$. All other properties are independent of the term language and provide general mechanisms to combine simpler properties into more complex ones, yielding a very general type system. In this chapter we do not formally distinguish between ordinary types and property types, though such a distinction has been useful in the study of refinement types [FP91, Fre94, Dav05a].

Our formulation of property types was fully explained and justified in Chapter 2 for a pure type assignment system; here, we focus on the bidirectionality of the rules. We do not extend the operational semantics: it is easiest to erase annotations before executing the program. Hence, type safety (Section 3.5.4) will follow directly from the result for the type assignment system (Theorem 2.21).

3.3.1 Intersections

A value v has type $A \wedge B$ if it has type A and type B. Because this is an introduction form, we proceed by *checking* v *against* A and B. Conversely, if e has type $A \wedge B$ then it must have both type A and type B, proceeding in the direction of synthesis.

$$\frac{\Gamma \vdash v \downarrow A \qquad \Gamma \vdash v \downarrow B}{\Gamma \vdash v \downarrow A \wedge B}\wedge\mathrm{I} \qquad \frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow A}\wedge\mathrm{E}_1 \quad \frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow B}\wedge\mathrm{E}_2$$

While these rules combine properties of the same term (and are therefore not an example of a Curry-Howard correspondence), the erasure of the terms still yields the ordinary logical rules for conjunction. Therefore, by the same reasoning as for ordinary types, the directionality of the rules follows from logical principles.

Usually, the elimination rules are a consequence of the subtyping rules (via the sub typing rule), but once bidirectionality is enforced, this is not the case and the rules must be taken as primitive. Note that the introduction form $\wedge$I is restricted to values because its general form for arbitrary expressions e is unsound in the presence of mutable references in call-by-value languages [DP00].

The subtyping rules are the same as in Chapter 2.

3.3.2 Greatest type: $\top$

A greatest type $\top$ can be thought of as the 0-ary form of intersection ($\wedge$). The rules are simply

$$\frac{\Gamma \vdash v \ \mathsf{ok}}{\Gamma \vdash v \downarrow \top}\top\mathrm{I} \qquad \frac{}{\Gamma \vdash A \leq \top}\top\mathrm{R}$$

There is no elimination or left subtyping rule for $\top$. Its typing rule is a 0-ary version of $\wedge$I, and the value restriction is also required (see Chapter 2).

3.3.3 Refined datatypes

As in the previous chapter, $\delta(i)$ is the type of values having datasort δ and index i. The rule for constructor application is

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \qquad \Gamma \vdash e \downarrow A}{\Gamma \vdash c(e) \downarrow \delta(i)}\delta\mathrm{I}$$

To derive $\Gamma \vdash$ **case** e **of** $ms \downarrow B$, we check that all the matches in ms check against B, under zero or more contexts appropriate to each arm. The case typing rules (Figure 3.3) are identical to the rules in the type assignment system (Figure 2.10, discussed in Section 2.3.4). except that $e : C$ and $ms :_B C$ have become $e \downarrow C$ and $ms \downarrow_B C$. As in the earlier system, when the case arm can be shown to be unreachable by virtue of the index refinements of the constructor type and the case subject, the assumptions added by rule δS-ct may be inconsistent ($\overline{\Gamma} \models \bot$). In order to skip such unreachable arms, we have rule contra:

$$\frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e \ \mathsf{ok}}{\Gamma \vdash e \downarrow A}\mathsf{contra}$$

This is neither an introduction nor an elimination rule, but we clearly cannot synthesize a type A out of nothing, so the only mode-correct formulation is one concluding $e \downarrow A$, which anyway matches the premise of δS-ct.

The typing rules for Π are

$$\frac{\Gamma, a{:}\gamma \vdash v \downarrow A}{\Gamma \vdash v \downarrow \Pi a{:}\gamma.\, A}\Pi\mathrm{I} \quad \frac{\Gamma \vdash e \uparrow \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e \uparrow [i/a]\, A}\Pi\mathrm{E}$$

$\boxed{\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e \downarrow C}$

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i' \vdash e \downarrow C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e \downarrow C}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta' \qquad \Gamma, x{:}A \vdash e\ \text{ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e \downarrow C}\ \delta\text{F-ct}$$

$$\frac{\begin{array}{c}\Gamma; c : A_1; c(x) : B \vdash e \downarrow C \\ \Gamma; c : A_2; c(x) : B \vdash e \downarrow C\end{array}}{\Gamma; c : A_1 \wedge A_2; c(x) : B \vdash e \downarrow C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A; c(x) : B \vdash e \downarrow C}{\Gamma; c : \Pi a{:}\gamma.\, A; c(x) : B \vdash e \downarrow C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A; c(x) : B \vdash e \downarrow C}{\Gamma; c : (P \supset A); c(x) : B \vdash e \downarrow C}\ \supset\text{-ct}$$

$\boxed{\Gamma \vdash ms \downarrow_B C}$

$$\frac{}{\Gamma \vdash \cdot \downarrow_B C}\ \text{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e \downarrow C \qquad \Gamma \vdash ms \downarrow_B C}{\Gamma \vdash (c(x) \Rightarrow e \mid ms) \downarrow_B C}\ \text{casearm}$$

Figure 3.3: Case typing rules in the simple tridirectional system

By our general assumption that contexts are well formed, the index variable a added to the context must be new, which can always be achieved via renaming. The directionality of these rules follows our general scheme. As with intersections, the introduction rule is restricted to values, to maintain type preservation in the presence of effects.

One potentially subtle issue with the introduction rule is that v cannot reference a in an internal type annotation, because that would violate α-conversion: one could not safely rename a to b in $\Pi a{:}\gamma.\, A$, which is the natural scope of a. We describe our solution, *contextual typing annotations*, in Section 3.4.

As written, in ΠL and ΠE we must guess the index i; in practice, we would plug in a new existentially quantified index variable and continue, using constraint solving to determine i. Thus, even if we had no existential types Σ in the system, the solver for the constraint domain would have to allow existentially quantified variables.

For the guarded type $P \supset A$, we again have the introduction rule check and the elimination rule synthesize:

$$\frac{\Gamma, P \vdash v \downarrow A}{\Gamma \vdash v \downarrow P \supset A}\ \supset\text{I} \qquad \frac{\Gamma \vdash e \uparrow P \supset A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e \uparrow A}\ \supset\text{E}$$

3.3.4 Indefinite property types

On values, the binary indefinite type is simply a union in the ordinary sense: if $v : A \vee B$ then either $v : A$ or $v : B$. The introduction rules directly express the simple logical interpretation, again using checking for the introduction form.

$$\frac{\Gamma \vdash e \downarrow A}{\Gamma \vdash e \downarrow A \vee B}\ \vee I_1 \qquad \frac{\Gamma \vdash e \downarrow B}{\Gamma \vdash e \downarrow A \vee B}\ \vee I_2$$

No restriction to values is needed for the introductions, but, dually to intersections, the elimination must be restricted. A sound formulation of the elimination rule in a type assignment form (Chapter 2) without a syntactic marker[3] requires an evaluation context $\mathcal{E}$ around the subterm of union type.

$$\frac{\Gamma \vdash e' : A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] : C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] : C\end{array}}{\Gamma \vdash \mathcal{E}[e'] : C}$$

This is where the "third direction" is necessary. We no longer move from terms to their immediate subterms, but when typechecking e we may have to decompose it into an evaluation context $\mathcal{E}$ and subterm e'. Using the analysis and synthesis judgments we have

$$\frac{\Gamma \vdash e' \uparrow A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] \downarrow C\end{array}}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\ \vee E$$

Here, if we can synthesize a union type for e'—which is in evaluation position in $\mathcal{E}[e']$—and check $\mathcal{E}[x]$ and $\mathcal{E}[y]$ against C, assuming that x and y have type A and type B respectively, we can conclude that $\mathcal{E}[e']$ checks against C. Note that the assumptions $x{:}A$ and $y{:}B$ can be read as $x{\uparrow}A$ and $y{\uparrow}B$ so we do indeed transition from $_ \uparrow A \vee B$ to $_ \uparrow A$ and $_ \uparrow B$. While typechecking still somehow follows the syntax, there may be many choices of $\mathcal{E}$ and e', leading to excessive nondeterminism (addressed in Chapter 5).

The 0-ary indefinite type is the empty or void type $\bot$; it has no values and therefore no introduction rules. For an elimination rule $\bot$E, we proceed by analogy with $\vee$E:

$$\frac{\Gamma \vdash e' \uparrow \bot \qquad \Gamma \vdash \mathcal{E}[e']\ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\ \bot E$$

As before, the expression must be an evaluation context $\mathcal{E}$ with e' in evaluation position.

For existential dependent types, the introduction rule presents no difficulties, and proceeds using the analysis judgment.

$$\frac{\Gamma \vdash e \downarrow [i/a]\,A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e \downarrow \Sigma a{:}\gamma.\,A}\ \Sigma I$$

For the elimination rule, we follow $\vee$E and $\bot$E:

$$\frac{\Gamma \vdash e' \uparrow \Sigma a{:}\gamma.\,A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] \downarrow C}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\ \Sigma E$$

As with Π, there is a potentially subtle issue: the index variable a must be new and cannot be mentioned in an annotation in $\mathcal{E}$.

[3]Pierce [Pie91b] used an explicit marker **case** e' **of** $x \Rightarrow e$ as the union elimination form. This is technically straightforward but a heavy burden on the programmer, particularly as markers would also be needed to eliminate Σ types, which are especially common in code without refinements; legacy code would have to be extensively "marked" to make it typecheck.

The bidirectional rules for the asserting type $P \otimes A$ are

$$\frac{\Gamma \vdash e \downarrow A \qquad \overline{\Gamma} \models P}{\Gamma \vdash e \downarrow P \otimes A}\otimes I \qquad \frac{\Gamma \vdash e' \uparrow P \otimes A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] \downarrow C}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\otimes E$$

3.3.5 Subtyping

The subtyping rules are unchanged from the previous chapter (Figure 2.11). We obtain decidability, reflexivity, and transitivity as before. Again, decidability relies on an ability to solve for existentially quantified variables; see Chapter 6.

3.3.6 The tridirectional rule

Considering rule $\bot$E to be the 0-ary version of $\vee$E for the binary indefinite type, what is the unary version? It is:

$$\frac{\Gamma \vdash e' \uparrow A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow C}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\,\mathsf{direct}$$

The type assignment version of this rule was admissible. However, due to the restriction to evaluation contexts, the bidirectional version is not. As a simple example, consider

$$\begin{array}{rcl} \textit{append} & : & \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(a+b) \\ \textit{filterpos} & : & \Pi n{:}\mathcal{N}.\, \mathsf{list}(n) \to \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ & \vdash & \textit{filterpos}\ [\ldots] \uparrow \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ \text{Goal:} & \vdash & \textit{append}\ [42]\ (\textit{filterpos}\ [\ldots]) \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k) \end{array}$$

where [42] is shorthand for $\mathsf{Cons}(42, \mathsf{Nil})$ and $[\ldots]$ is some literal list. Without rule direct we cannot derive the goal, because we cannot introduce the k on the type being checked against. To do so, we would need to introduce the index variable m representing the length of the list returned by *filterpos* $[\ldots]$, and use $m+1$ for k. But *filterpos* $[\ldots]$ is not in evaluation position, because *append* [42] must be evaluated first. However, *append* [42] synthesizes only type $\Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(1+b)$. We would be stuck without rule direct, which enables us to reduce

$$\vdash \textit{append}\ [42]\ (\textit{filterpos}\ [\ldots]) \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k)$$

to

$$x : \Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(1+b) \vdash x\ (\textit{filterpos}\ [\ldots]) \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k)$$

Since x is a value, (*filterpos* $[\ldots]$) is in evaluation position and we can use the existential elimination rule

$$x{:}\Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(1+b), m{:}\mathcal{N}, y{:}\mathsf{list}(m) \vdash x\ y \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k)$$

Now we can complete the derivation with ΣI, using $1+m$ for k and several straightforward steps.

3.4 Contextual typing annotations

Our tridirectional system so far has the property that only terms in normal form have types. Terms not in normal form, such as $(\lambda x.\, x)()$, neither synthesize nor check against a type. This is because the function part of an application must synthesize a type, but there is no rule for $\lambda x.\, e$ that has a synthesizing conclusion.

Clearly, we need some form of annotation, but this is not as straightforward as one might hope. In our setting, two issues arise: checking against intersections, and index variable scoping.

3.4.1 Checking against intersections

Consider the following function, which conses 42 to its argument.

$$\mathit{cons42} = \big(\lambda x.\, (\lambda y.\, \mathsf{Cons}(42, x))()\big) : (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$$

This does not typecheck: $\lambda y.\, \mathsf{Cons}(42, x)$ needs an annotation. Observe that by rule $\wedge$I, *cons42* will be checked twice: first against $\mathsf{odd} \to \mathsf{even}$, then against $\mathsf{even} \to \mathsf{odd}$. Hence, we cannot write $(\lambda y.\, \mathsf{Cons}(42, x)) : (\mathbf{1} \to \mathsf{even})$—it is correct only when checking *cons42* against $\mathsf{odd} \to \mathsf{even}$. Moreover, we cannot write

$$(\lambda y.\, \mathsf{Cons}(42, x)) : (\mathbf{1} \to \mathsf{even}) \wedge (\mathbf{1} \to \mathsf{odd})$$

We need to use $\mathbf{1} \to \mathsf{even}$ while checking *cons42* against $\mathsf{odd} \to \mathsf{even}$, and $\mathbf{1} \to \mathsf{odd}$ while checking *cons42* against $\mathsf{even} \to \mathsf{odd}$. Exasperatingly, union types are no help here: $(\lambda y.\, \mathsf{Cons}(42, x)) : (\mathbf{1} \to \mathsf{even}) \vee (\mathbf{1} \to \mathsf{odd})$ is a value of type $\mathbf{1} \to \mathsf{even}$ or of type $\mathbf{1} \to \mathsf{odd}$, but we do not know which; following $\vee$E, we must suppose it has type $\mathbf{1} \to \mathsf{even}$ and then check its application to $\mathbf{1}$, *and then* suppose it has type $\mathbf{1} \to \mathsf{odd}$ and check its application to $\mathbf{1}$. Only one of these checks will succeed—a different one, depending on which conjunct of $(\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd})$ we happen to be checking *cons42* against—but according to $\vee$E both need to succeed.

Reynolds' Forsythe language [Rey88] addressed this problem by allowing function arguments to be annotated with a list of alternative types; the typechecker guesses the right one, backtracking if necessary. Pierce extended this approach, allowing the alternative to be named [Pie91a, pp. 21, 31] (a generalization Reynolds incorporated in the revised Forsythe report [Rey96]). Davies took a closely similar approach in his datasort refinement checker [Dav05a], allowing a term to be annotated with $(e : A, B, \ldots)$. In that notation, the above function could be written as

$$\begin{aligned} \mathit{cons42} \;=\; & \big(\lambda x.\, (((\lambda y.\, \mathsf{Cons}(42, x)) : \mathbf{1} \to \mathsf{even}, \mathbf{1} \to \mathsf{odd})()\big) \\ & : (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd}) \end{aligned}$$

Now the typechecker can choose $\mathbf{1} \to \mathsf{even}$ when checking against $\mathbf{1} \to \mathsf{odd}$. This notation is easy to use and effective but introduces additional backtracking, since the typechecker must guess which type to use.

3.4.2 Index variable scoping

Some functions need type annotations inside their bodies, such as this (contorted) identity function on lists.

$$id \;=\; \big(\lambda x.\,(\lambda z.\,x)\,()\big) \;:\; \Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \rightarrow \mathsf{list}(a)$$

In a bidirectional system, the function part of an application must synthesize a type, but we have no rule to synthesize a type for a λ-abstraction. So we need an annotation on $(\lambda z.\,x)$. We need to show that the whole application checks against $\mathsf{list}(a)$, so we might try

$$(\lambda z.\,x) \;:\; \mathbf{1} \rightarrow \mathsf{list}(a)$$

But this would violate variable scoping. α-convertibility dictates that $\Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \rightarrow \mathsf{list}(a)$ and $\Pi b{:}\mathcal{N}.\,\mathsf{list}(b) \rightarrow \mathsf{list}(b)$ must be indistinguishable, which would be violated if we permitted

$$\lambda x.\,((\lambda z.\,x) : \mathbf{1} \rightarrow \mathsf{list}(a))\,() \;\downarrow\; \Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \rightarrow \mathsf{list}(a)$$

but not

$$\lambda x.\,((\lambda z.\,x) : \mathbf{1} \rightarrow \mathsf{list}(a))\,() \;\downarrow\; \Pi b{:}\mathcal{N}.\,\mathsf{list}(b) \rightarrow \mathsf{list}(b)$$

Xi already noticed this problem and introduced a term-level abstraction over index variables, $\Lambda a.e$, to mirror universal index quantification $\Pi a{:}\gamma.\,A$ [Xi98]. But this violates the basic principle of property types that the term should not change as property types are added, and fails in the presence of intersections. For example, we would expect the reverse function on lists, *rev*, to satisfy

$$rev \;:\; \underbrace{(\Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \rightarrow \mathsf{list}(a))}_{\text{one outside quantifier}} \;\wedge\; \underbrace{((\Sigma b{:}\mathcal{N}.\,\mathsf{list}(b)) \rightarrow \Sigma c{:}\mathcal{N}.\,\mathsf{list}(c))}_{\text{no outside quantifiers}}$$

but the first component of the intersection would demand a term-level index abstraction, while the second would not tolerate one.

3.4.3 Contextual subtyping

We address these two problems by a method that extends and improves the notation of comma-separated alternatives. The essential idea is to allow a context to appear in the annotation along with each type:

$$\begin{array}{lrcl} \text{Typings} & As & ::= & \Gamma \vdash A \mid \Gamma \vdash A,\, As \\ \text{Terms} & e & ::= & \ldots \mid (e : As) \end{array}$$

where in $(e : \Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n)$ each context Γ_k contains types for a subset of the free program variables in e, as well as index variable sorts for index variables used in Γ_k and A_k. The scope of Γ_k is just A_k; program variable typings appearing in Γ_k refer to program variables declared outside the annotated term, while index variable sortings are binders whose scope is limited to the rest of Γ_k (since index variables must appear to the left of their use in a context) and A_k. Index variables

can appear in a term e only in a type annotation, and they must be bound by some Γ_k within that annotation. Thus, no term has any free index variables.

In the first approximation we can think of such an annotated term as follows: if $\Gamma_k \vdash e \downarrow A_k$ then $\Gamma \vdash (e : \Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n) \uparrow A_k$ if the current assumptions in Γ validate the assumptions in Γ_k. For example,the second judgment below is not derivable, since $x{:}\mathsf{odd}$ does not validate $x{:}\mathsf{even}$ (because $\mathsf{odd} \not\leq \mathsf{even}$).

$$\begin{aligned} x{:}\mathsf{even} &\vdash \big((\lambda y.\,\mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ &\qquad\qquad\qquad\qquad\quad x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}\big) \uparrow \mathbf{1} \to \mathsf{odd} \\ x{:}\mathsf{odd} &\nvdash \big((\lambda y.\,\mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ &\qquad\qquad\qquad\qquad\quad x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}\big) \uparrow \mathbf{1} \to \mathsf{odd} \end{aligned}$$

In practice, this should significantly reduce the nondeterminism associated with type annotations in the presence of intersection. However, we still need to generalize the rule in order to correctly handle index variable scoping.

Returning to our earlier example, we would like to find an annotation As allowing us to derive

$$\vdash \lambda x.\, \big((\lambda z.\, x) : As\big)\,() \;\downarrow\; \Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \to \mathsf{list}(a)$$

The idea is to use a *locally* declared index variable (here, b)

$$\lambda x.\, \big((\lambda z.\, x) : (b{:}\mathcal{N}, x{:}\mathsf{list}(b) \vdash \mathbf{1} \to \mathsf{list}(b))\big)$$

to make the typing annotation self-contained. Now, when we check if the current assumptions for x validate local assumption for x, we are permitted to instantiate b to any index i. In this example, we could substitute a for b. As a result, we end up checking $(\lambda z.\, x) \downarrow \mathbf{1} \to \mathsf{list}(a)$, even though the annotation does not mention a. Note that in an annotation $e : (\Gamma_0 \vdash A_0), As$, all index variables declared in Γ_0 are considered bound and can be renamed consistently in Γ_0 and A_0. In contrast, free term variables in Γ_0, such as x, may actually occur in e and so cannot be renamed freely.

These considerations lead us to a *contextual subtyping* relation $\lesssim$:

$$(\Gamma_0 \vdash A_0) \;\lesssim\; (\Gamma \vdash A)$$

which is contravariant in the contexts Γ_0 and Γ. It would be covariant in A_0 and A, except that in the way it is invoked, Γ_0, A_0, and Γ are known and A is generated as an instance of A_0. This should become more clear when we consider its use in the new typing rule

$$\frac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \downarrow A}{\Gamma \vdash (e : (\Gamma_0 \vdash A_0), As) \uparrow A}\;\textsf{ctx-anno}$$

where we regard the typings as unordered (so $\Gamma_0 \vdash A_0$ could occur anywhere in the list). In the bidirectional style, Γ, e, Γ_0, A_0 and As are known when we try this rule. While finding a derivation of $(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$ we generate A, which is the synthesized type of the original annotated expression e, if in fact e checks against A. It is also possible that $(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$ fails to have a derivation when Γ_0 and Γ have incompatible declarations for the term variables occurring in

$$\begin{aligned}
\text{Typings}\ \ As &::= \Gamma \vdash A \ \mid\ \Gamma \vdash A,\ As \\
\text{Terms}\ \ e &::= \ldots \mid (e : As) \\
\text{Values}\ \ v &::= \ldots \mid (v : As) \\
\text{Eval. contexts}\ \ \mathcal{E} &::= \ldots \mid (\mathcal{E} : As)
\end{aligned}$$

Figure 3.4: Language additions for contextual typing annotations

$$\frac{}{(\cdot \vdash A) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-empty}$$

$$\frac{\overline{\Gamma} \vdash i : \gamma_0 \qquad ([i/a]\,\Gamma_0 \vdash [i/a]\,A_0) \lesssim (\Gamma \vdash A)}{(a{:}\gamma_0, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-ivar}$$

$$\frac{\overline{\Gamma} \models P \qquad (\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}{(P, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-prop}$$

$$\frac{\Gamma \vdash \Gamma(x) \leq B_0 \qquad (\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}{(x{:}B_0, \Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)}\ \lesssim\text{-pvar}$$

Figure 3.5: Contextual subtyping

them (as in the example above, if $\Gamma_0 = x{:}\mathsf{odd}$ and $\Gamma = \ldots, x{:}\mathsf{even}, \ldots$), in which case we try another typing $(\Gamma_k \vdash A_k)$.

The formal rules for contextual subtyping are given in Figure 3.5. Besides the considerations above, we also must make sure that any possible assumptions P about the index variables in Γ_0 are indeed entailed by the current context, after any possible substitution has been applied (this is why we traverse Γ_0 from left to right).

While the examples above are artificial, similar situations arise in ordinary programs in the common situation when local function definitions reference free variables. Two small examples of this kind are given in Figure 3.6, presented in the style of ML, where we follow the tradition of systems such as Davies' and write typing annotations inside bracketed comments.

The essence of the completeness result we prove in Section 3.4.5 is that annotations can be added to any term that is well typed in the type assignment system to yield a well typed term in the tridirectional system. For this result to hold, $\lesssim$ must be reflexive:

$$(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$$

Reflexivity (together with proper α-conversion) is *sufficient* for completeness: in the proof of completeness, where we see $\Gamma \vdash e : A$ we can simply add an annotation $(\Gamma \vdash A)$. But it would be absurd to make programmers type in entire contexts—not only is the length impractical, but whenever a declaration is added every contextual annotation in its scope would have to be changed!

Reflexivity of $\lesssim$ follows easily from the following lemma.

Lemma 3.1. $(\Gamma_2 \vdash A) \lesssim (\Gamma_1, \Gamma_2 \vdash A)$.

Proof. By induction on Γ_2.

1. Case: $\Gamma_2 = \cdot$. The result follows by $\lesssim$-empty.

2. Case: $\Gamma_2 = x{:}B, \Gamma$.

$$\begin{array}{rl} \Gamma_1, x{:}B, \Gamma \vdash B \leq B & \text{By reflexivity of subtyping} \\ (\Gamma \vdash A) \lesssim (\Gamma_1, x{:}B, \Gamma \vdash A) & \text{By IH} \\ (x{:}B, \Gamma \vdash A) \lesssim (\Gamma_1, x{:}B, \Gamma \vdash A) & \text{By } \lesssim\text{-pvar} \end{array}$$

3. Case: $\Gamma_2 = a{:}\gamma, \Gamma$.

$$\begin{array}{rl} \overline{\Gamma_1, a{:}\gamma, \Gamma} \vdash a : \gamma & \text{Using assumption} \\ (\Gamma \vdash A) \lesssim (\Gamma_1, a{:}\gamma, \Gamma \vdash A) & \text{By IH} \\ ([a/a]\ \Gamma \vdash [a/a]\ A) \lesssim (\Gamma_1, a{:}\gamma, \Gamma \vdash A) & \text{Identity subst.} \\ (a{:}\gamma, \Gamma \vdash A) \lesssim (\Gamma_1, a{:}\gamma, \Gamma \vdash A) & \text{By } \lesssim\text{-ivar} \end{array}$$

4. Case: $\Gamma_2 = P, \Gamma$.

$$\begin{array}{rl} \overline{\Gamma_1, P, \Gamma} \models P & \text{Using assumption} \\ (\Gamma \vdash A) \lesssim (\Gamma_1, P, \Gamma \vdash A) & \text{By IH} \\ (P, \Gamma \vdash A) \lesssim (\Gamma_1, P, \Gamma \vdash A) & \text{By } \lesssim\text{-prop} \end{array}$$

□

Corollary 3.2 (Reflexivity). $(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$.

Proof. By Lemma 3.1 with Γ_2 empty. □

3.4.4 Soundness

Let $|e|$ denote the erasure of all typing annotations from e.

Theorem 3.3 (Soundness, Tridirectional). *If* $\Gamma \vdash e \uparrow A$ *or* $\Gamma \vdash e \downarrow A$ *then* $\Gamma \vdash |e| : A$.

Proof. By straightforward induction on the derivation. Except for ctx-anno, which is not in the type assignment system, the type assignment system is precisely the tridirectional system with : replacing $\uparrow$ and $\downarrow$, so all cases except ctx-anno are utterly straightforward. For ctx-anno on a term $(e_0 : A)$, apply the IH to obtain $\Gamma \vdash |e_0| : A$ and use the equivalence $|e_0| = |(e_0 : As)|$. □

3.4.5 Completeness

We cannot just take a derivation $\Gamma \vdash e : A$ in the type assignment system and obtain a derivation $\Gamma \vdash e \uparrow A$ in the tridirectional system. For example, $\vdash \lambda x.\, x : A \to A$ for any type A, but in the tridirectional system $\lambda x.\, x$ does not synthesize a type. However, if we add a typing annotation, we can derive

$$\vdash (\lambda x.\, x : (\vdash A \to A)) \uparrow A \to A$$

Clearly, the completeness result must be along the lines of "If $\Gamma \vdash e : A$, then there is an annotated version e' of e such that $\Gamma \vdash e' \uparrow A$." To formulate this result (Corollary 3.16, a special case of Theorem 3.15) we need a few definitions and lemmas.

```
true ≼ bool,  false ≼ bool              even ≼ nat,  odd ≼ nat
evenlist ≼ list,  oddlist ≼ list
𝒮(Nil) = evenlist
𝒮(Cons) = (evenlist → oddlist) ∧ (oddlist → evenlist)
eq : (even * odd → false)
    ∧ (odd * even → false)
    ∧ (nat * nat → bool)
(*[  member :  (even * oddlist → false)
              ∧ (odd * evenlist → false)
              ∧ (nat * list → bool)  ]*)
fun member (x, xs) =
  (*[  mem : x:even ⊢ (evenlist → bool) ∧ (oddlist → false),
             x:odd ⊢ (evenlist → false) ∧ (oddlist → bool),
             x:nat ⊢ list → bool  ]*)
  let fun mem xs =
     case xs of Nil ⇒ False
      | Cons(y, ys) ⇒ eq(x, y) orelse mem ys
  in mem xs end


𝒮(Nil) = list(0)
𝒮(Cons) = Πa:𝒩. int * list(a) → list(a + 1)
(*[  append : Πa:𝒩. Πb:𝒩. list(a) * list(b) → list(a + b)  ]*)
fun append (xs, ys) =
  (*[  app : c:𝒩, ys:list(c) ⊢ Πa:𝒩. list(a) → list(a + c)  ]*)
  let fun app xs = case xs of Nil ⇒ ys
      | Cons(x, xs) ⇒ Cons(x, app xs)
  in app xs end
```

Figure 3.6: Examples of contextual annotations

Definition 3.4. A term is in *synthesizing form* if it has any of the forms

$$x, \quad e_1 e_2, \quad u, \quad (e : As), \quad \mathbf{fst}(e), \quad \mathbf{snd}(e)$$

Proposition 3.5. *If* $\Gamma \vdash e \uparrow A$ *then* e *is in synthesizing form.*

Proof. By induction on the derivation. □

Remark 3.6. The point of a typing annotation is to obtain a synthesizing term from a non-synthesizing one. It is never necessary to add an annotation to the root of a term in synthesizing form (though annotations may be needed *inside* the term).

Definition 3.7. e' *extends* a term e, written $e' \sqsupseteq e$ iff e' is e with zero or more additional typing annotations and e' has no subterm $(e'' : As)$ where e'' is in synthesizing form.

Definition 3.8. e' *lightly extends* a term e, written $e' \sqsupseteq_\ell e$ iff e' is e with zero or more typing annotations added to lists of typing annotations *already present* in e. For example, $(e : As, \Gamma \vdash A')$ lightly extends $(e : As)$, but $(e : \Gamma \vdash A')$ does not lightly extend e.

Proposition 3.9. $\sqsupseteq$ *and* $\sqsupseteq_\ell$ *are reflexive and transitive.*

Proof. Obvious from the definitions. □

Lemma 3.10. *If* e value *and* $e' \sqsupseteq e$ *then* e' value.

Proof. By a straightforward induction on e' (making use of $(v : As)$ value). □

Lemma 3.11. *If* $e' \sqsupseteq \mathcal{E}[e_0]$ *where* e_0 *is in synthesizing form then there exist* $\mathcal{E}'$ *and* e_0' *such that* $e' = \mathcal{E}'[e_0']$ *and* $e_0' \sqsupseteq e_0$ *and for all terms* e_1 *in synthesizing form it is the case that* $\mathcal{E}'[e_1] \sqsupseteq \mathcal{E}[e_1]$.

Proof. By induction on $\mathcal{E}$. We show two representative cases.

- $\boxed{\mathcal{E} = []}$

  Let $\mathcal{E}' = []$ and $e_0' = e'$. Then $e' = \mathcal{E}'[e_0']$. Since $e_0' = e'$ and $\mathcal{E}[e_0] = e_0$ the desired $e_0' \sqsupseteq e_0'$ follows from $e' \sqsupseteq \mathcal{E}[e_0]$. Since $\mathcal{E}' = \mathcal{E}$, $\mathcal{E}'[e_1] \sqsupseteq \mathcal{E}[e_1]$ follows by Proposition 3.9.

- $\boxed{\mathcal{E} = (v, \mathcal{E}_*)}$

  By $e' \sqsupseteq (v, \mathcal{E}_*[e_0])$ and Definition 3.7, either $e' = (v', e_*')$ or $e' = ((v', e_*') : As)$, where $v' \sqsupseteq v$ and $e_*' \sqsupseteq \mathcal{E}_*[e_0]$. We assume the former case, $e' = (v', e_*')$; the latter case is similar. By IH, there exist $\mathcal{E}_*'$ and e_0' such that

  $$e_*' = \mathcal{E}_*'[e_0'] \text{ and } e_0' \sqsupseteq e_0 \text{ and for all } e_1, \;\; \mathcal{E}_*'[e_1] \sqsupseteq \mathcal{E}_*[e_1]\,.$$

  Let $\mathcal{E}' = (v', \mathcal{E}_*')$ (we know by Lemma 3.10 that v' is a value, so $\mathcal{E}'$ is indeed an evaluation context). Then $(v', \mathcal{E}_*'[e_0']) = \mathcal{E}'[e_0']$. Since we have $e' = (v', e_*')$ and $e_*' = \mathcal{E}_*'[e_0']$, we can conclude $e' = \mathcal{E}'[\mathcal{E}_0']$, the first part of what was to be shown. The second part, $e_0' \sqsupseteq e_0$, we already know by IH. Now take any e_1. We know $v' \sqsupseteq v$ and by IH we have $\mathcal{E}_*'[e_1] \sqsupseteq \mathcal{E}_*[e_1]$. Therefore $(v', \mathcal{E}_*'[e_1]) \sqsupseteq (v, \mathcal{E}_*[e_1])$. By $\mathcal{E}'[e_1] = (v', \mathcal{E}_*'[e_1])$ and $(v, \mathcal{E}_*[e_1]) = \mathcal{E}[e_1]$,

  $$\mathcal{E}'[e_1] \sqsupseteq \mathcal{E}[e_1]$$

  which is the last part of what was to be shown. □

Lemma 3.12. *If $e' \sqsupseteq_\ell \mathcal{E}[e_0]$ where e_0 is in synthesizing form then there exist $\mathcal{E}'$ and e'_0 such that $e' = \mathcal{E}'[e'_0]$ and $e'_0 \sqsupseteq_\ell e_0$ and for all e_1 in synthesizing form it is the case that $\mathcal{E}'[e_1] \sqsupseteq_\ell \mathcal{E}[e_1]$.*

Proof. By induction on $\mathcal{E}$, following the proof of Lemma 3.11. □

To reduce verbosity, when a result has two parts that are identical except for the direction ($\uparrow$ or $\downarrow$) of the judgment(s) involved, we write $\underline{\downarrow\uparrow}$ to represent both. If there is more than one judgment mentioned, the directions must be read consistently: either as all $\uparrow$, or all $\downarrow$. Thus, Lemma 3.13 (just below) is equivalent to the verbose statement:

If $e' \sqsupseteq_\ell e$ then

(1) If $\Gamma \vdash e \downarrow A$ then $\Gamma \vdash e' \downarrow A$.

(2) If $\Gamma \vdash e \uparrow A$ then $\Gamma \vdash e' \uparrow A$.

Lemma 3.13 (Light Extension)**.** *If $e' \sqsupseteq_\ell e$ and $\Gamma \vdash e \underline{\downarrow\uparrow} A$ then $\Gamma \vdash e' \underline{\downarrow\uparrow} A$. (Similarly for case typing judgments and matches ms.)*

Proof. By induction on the derivation of the typing judgment. All cases are straightforward: either e and e' must be identical (for instance, for **1**I), or we apply the IH to all premises, which leads directly to the result. □

Recall that the rule $\wedge$I led to the need for more than one typing annotation on a term. It should be no surprise, then, that the $\wedge$I case in the completeness proof is interesting. Applying the induction hypothesis to each premise $v : A$, $v : B$ yields two possibly *different* annotated terms v'_A and v'_B such that $v'_A \downarrow A$ and $v'_B \downarrow B$. But given a notion of *monotonicity* under annotation, we can incorporate both annotations into a single v' such that $v' \downarrow A$ and $v' \downarrow B$. However, the obvious formulation of monotonicity

If $e \downarrow A$ and $e' \sqsupseteq e$ then $e' \downarrow A$

does not hold: given a list of annotations As the type system must use at least one of them—it cannot ignore them all. Thus $\vdash ((): (\vdash \top)) \downarrow \mathbf{1}$ is not derivable, even though $\vdash () \downarrow \mathbf{1}$ is derivable and $((): (\vdash \top)) \sqsupseteq ()$. However, further annotating $((): (\vdash \top))$ to $((): (\vdash \top), (\vdash \mathbf{1}))$ yields a term that checks against both $\top$ and **1**. Note that this further annotation was light—we added a typing to an existing annotation. This observation leads to Lemma 3.14.

Lemma 3.14 (Monotonicity under annotation)**.**

(1) If $\Gamma \vdash e \downarrow A$ and $e' \sqsupseteq e$ then there exists $e'' \sqsupseteq_\ell e'$ such that $\Gamma \vdash e'' \downarrow A$.

(2) If $\Gamma \vdash e \uparrow A$ and $e' \sqsupseteq e$ then there exists $e'' \sqsupseteq_\ell e'$ such that $\Gamma \vdash e'' \uparrow A$.

(3) If $\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e \downarrow C$ and $e' \sqsupseteq e$ then there exists $e'' \sqsupseteq_\ell e'$ such that $\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e'' \downarrow C$.

(4) If $\Gamma \vdash ms \downarrow_B C$ and $ms' \sqsupseteq ms$ then there exists $ms'' \sqsupseteq_\ell ms'$ such that $\Gamma \vdash ms'' \downarrow_B C$.

Proof. By induction on the derivation of the typing judgment.

The cases for rules such as $\wedge$I, $\wedge$-ct and sub, which have the same term in the premise(s) and conclusion, are straightforward (for rules such as $\wedge$I and ΠI where the term must be a value, we use Lemma 3.10). For contra, just reapply the rule. We show several representative cases of the remaining rules.

1. **Case** var: $\mathcal{D} :: \dfrac{\Gamma(x) = A}{\Gamma \vdash x \uparrow A}$

   By the definition of $\sqsupseteq$, e' can contain no annotations on the roots of terms in synthesizing form such as x. Therefore $e' = x$, and we already have $\Gamma \vdash x \uparrow A$. By Proposition 3.9, $x \sqsupseteq_\ell x$.

2. **Case** fixvar: Similar to the preceding case.

3. **Case** $\to$I: $\mathcal{D} :: \dfrac{\Gamma, x{:}A_1 \vdash e_0 \downarrow A_2}{\Gamma \vdash \lambda x.\, e_0 \downarrow A_1 \to A_2}$

   We have $e' \sqsupseteq \lambda x.\, e_0$. Therefore there exists some subterm e_0' of e' such that $e_0' \sqsupseteq e_0$.

   $\Gamma, x{:}A_1 \vdash e_0 \downarrow A_2$ — Subd.
   $\Gamma, x{:}A_1 \vdash e_0'' \downarrow A_2,\ e_0'' \sqsupseteq_\ell e_0'$ — By IH
   $\Gamma \vdash \lambda x.\, e_0'' \downarrow A_1 \to A_2$ — By $\to$I
   $\lambda x.\, e_0'' \sqsupseteq_\ell \lambda x.\, e_0'$ — By Definition 3.8

   If e' has no annotation at its root, we are done. If it does, we have $e' = (\lambda x.\, e_0' : As)$. By Corollary 3.2, $(\Gamma \vdash A_1 \to A_2) \lesssim (\Gamma \vdash A_1 \to A_2)$. By ctx-anno,

   $$\Gamma \vdash ((\lambda x.\, e_0'') : As, \Gamma \vdash A_1 {\to} A_2) \downarrow A_1 \to A_2$$

   By $\lambda x.\, e_0'' \sqsupseteq_\ell \lambda x.\, e_0'$ and Definition 3.8,

   $$((\lambda x.\, e_0'') : As, (\Gamma \vdash A_1 {\to} A_2)) \sqsupseteq_\ell (\lambda x.\, e_0' : As)$$

4. **Case** fix, $*$I, $*$E$_1$, $*$E$_2$, δI, δE: Similar to the preceding case.

5. **Case** $\to$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e_1 \uparrow B \to A \qquad \Gamma \vdash e_2 \downarrow B}{\Gamma \vdash e_1 e_2 \uparrow A}$

   $e' \sqsupseteq e_1 e_2$ — Given
   $\exists e_1', e_2'.\ \ e' = e_1' e_2', e_1' \sqsupseteq e_1, e_2' \sqsupseteq e_2$ — By Definition 3.7
   $\Gamma \vdash e_1 \uparrow B \to A$ — Subd.
   $\Gamma \vdash e_1'' \uparrow B \to A,\ e_1'' \sqsupseteq_\ell e_1'$ — By IH
   $\Gamma \vdash e_2 \downarrow B$ — Subd.
   $\Gamma \vdash e_2'' \downarrow B,\ e_2'' \sqsupseteq_\ell e_2'$ — By IH
   $\Gamma \vdash e_1'' e_2'' \uparrow A$ — By $\to$E
   $e_1'' e_2'' \sqsupseteq_\ell e_1' e_2'$ — By Definition 3.8

6. **Case** $\vee$E:

$$\mathcal{D} :: \dfrac{\Gamma \vdash e_0 \uparrow B_1 \vee B_2 \quad \begin{array}{c}\Gamma, x{:}B_1 \vdash \mathcal{E}[x] \downarrow A \\ \Gamma, y{:}B_2 \vdash \mathcal{E}[y] \downarrow A\end{array}}{\Gamma \vdash \mathcal{E}[e_0] \downarrow A} \vee E$$

By Proposition 3.5, e_0 is in synthesizing form. By Lemma 3.11, there exist $\mathcal{E}'$ and e_0' such that $e' = \mathcal{E}'[e_0']$ and $e_0' \sqsupseteq e_0$ and $\mathcal{E}'[x] \sqsupseteq \mathcal{E}[x]$.

$\Gamma \vdash e_0 \uparrow B_1 \vee B_2$	Subd.
$e_0' \sqsupseteq e_0$	Above
$\Gamma \vdash e_0'' \uparrow B_1 \vee B_2,\ e_0'' \sqsupseteq_\ell e_0'$	By IH
$\Gamma, x{:}B_1 \vdash \mathcal{E}[x] \downarrow A$	Subd.
$\mathcal{E}'[x] \sqsupseteq \mathcal{E}[x]$	Above
$\Gamma, x{:}B_1 \vdash e'' \downarrow A,\ e'' \sqsupseteq_\ell \mathcal{E}'[x]$	By IH
$\exists \mathcal{E}''.\ \ e'' = \mathcal{E}''[x],\ \mathcal{E}''[y] \sqsupseteq_\ell \mathcal{E}[y]$	By Lemma 3.12
$\Gamma, y{:}B_2 \vdash \mathcal{E}[y] \downarrow A$	Subd.
$\Gamma, y{:}B_2 \vdash e''' \downarrow A,\ e''' \sqsupseteq_\ell \mathcal{E}''[y]$	By IH
$\exists \mathcal{E}'''.\ \ e''' = \mathcal{E}'''[y],\ \mathcal{E}'''[x] \sqsupseteq_\ell \mathcal{E}''[x]$	By Lemma 3.12
$\Gamma, y{:}B_2 \vdash \mathcal{E}'''[y] \downarrow A$	By $e''' = \mathcal{E}'''[y]$ and $\Gamma, y{:}B_2 \vdash e''' \downarrow A$
$\Gamma, x{:}B_1 \vdash \mathcal{E}'''[x] \downarrow A$	By Lemma 3.13
$\Gamma \vdash \mathcal{E}'''[e_0''] \downarrow A$	By $\vee$E
$\mathcal{E}'''[e_0''] \sqsupseteq_\ell \mathcal{E}'[e_0']$	By Definition 3.8

7. **Case** direct, $\bot$E, ΣE, $\otimes$E: Similar to the $\vee$E case, but simpler.

8. **Case 1**I:

$$\mathcal{D} :: \dfrac{}{\Gamma \vdash () \downarrow \mathbf{1}}$$

We have $e' \sqsupseteq ()$. By definition of $\sqsupseteq$, either $e' = ()$ or $e' = (() : As)$.

$\Gamma \vdash () \downarrow \mathbf{1}$ By **1**I

In the first case ($e' = ()$) we're done. In the second case:

$(\Gamma \vdash \mathbf{1}) \lesssim (\Gamma \vdash \mathbf{1})$	By Corollary 3.2
$\Gamma \vdash (() : As, (\Gamma \vdash \mathbf{1})) \uparrow \mathbf{1}$	By ctx-anno
$\Gamma \vdash \mathbf{1} \leq \mathbf{1}$	By reflexivity of $\leq$
$\Gamma \vdash (() : As, (\Gamma \vdash \mathbf{1})) \downarrow \mathbf{1}$	By sub

9. **Case** ctx-anno:

$$\mathcal{D} :: \dfrac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \quad \Gamma \vdash e_0 \downarrow A}{\Gamma \vdash (e_0 : (\Gamma_0 \vdash A_0), As) \uparrow A}$$

We have $e' \sqsupseteq (e_0 : (\Gamma_0 \vdash A_0), As)$. By definition of $\sqsupseteq$, $e' = (e_0' : (\Gamma_0 \vdash A_0), As, Bs)$ where $e_0' \sqsupseteq e_0$.

$\Gamma \vdash e_0 \downarrow A$	Subd.
$\Gamma \vdash e_0'' \downarrow A,\ e_0'' \sqsupseteq_\ell e_0'$	By IH
$(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$	Subd.
$\Gamma \vdash (e_0'' : (\Gamma_0 \vdash A_0), As, Bs) \uparrow A$	By ctx-anno

By definition of $\sqsupseteq_\ell$, we get $(e_0'' : (\Gamma_0 \vdash A_0), As, Bs) \sqsupseteq_\ell (e_0' : (\Gamma_0 \vdash A_0), As, Bs)$.

10. **Case** $\top$I: $\mathcal{D} :: \dfrac{\Gamma \vdash v \text{ ok}}{\Gamma \vdash v \downarrow \top}$

    We have $e' \sqsupseteq_\ell v$, so by Lemma 3.10, e' value.

$\Gamma \vdash e'$ ok	Follows from $\Gamma \vdash v$ ok
$\Gamma \vdash e' \downarrow \top$	By $\top$I
$e' \sqsupseteq_\ell e'$	By Proposition 3.9

□

Theorem 3.15 (Completeness, Tridirectional). *If* $\Gamma \vdash e : A$ *and* $e' \sqsupseteq e$ *then*

(i) there exists e_1'' *such that* $e_1'' \sqsupseteq e'$ *and* $\Gamma \vdash e_1'' \downarrow A$

(ii) there exists e_2'' *such that* $e_2'' \sqsupseteq e'$ *and* $\Gamma \vdash e_2'' \uparrow A$

If $\Gamma; c : A^{con}; c(x) : B \vdash e : C$ *and* $e' \sqsupseteq e$ *then there exists* e'' *such that* $e'' \sqsupseteq e'$ *and* $\Gamma; c : A^{con}; c(x) : B \vdash e'' \downarrow C$.

If $\Gamma \vdash ms :_B C$ *and* $ms' \sqsupseteq ms$ *then there exists* ms'' *such that* $ms'' \sqsupseteq ms'$ *and* $\Gamma \vdash ms'' \downarrow_B C$.

Proof. By induction on the given derivation.

The presence of e' (resp. ms') is motivated by the need to traverse multiple subderivations, adding annotations at each one. The simplest instance is in the $\wedge$I case.

Whenever e is not in synthesizing form, it is possible that e has no annotation at its root, while e' does. The extra steps to handle this situation are all similar, so except in $\rightarrow$I, the first such case, we assume e' has no annotation at its root.

Except in the **1**I case, one can prove either part (i) or part (ii) and then prove the other part in a manner that is similar across cases: In cases where part (i) is most naturally proved first, part (ii) follows by adding a type annotation to e and applying ctx-anno. In cases where part (ii) is naturally shown first, part (i) follows by applying sub using the reflexivity of subtyping. Except in the $\rightarrow$I, $\rightarrow$E and **1**I cases, we elide these steps.

The case-typing and matches-typing parts of the theorem are completely straightforward, with the exception of the $\wedge$-ct case, in which the multiple premises typing the same term necessitate reasoning following that in the $\wedge$I case below.

1. **Case** $\rightarrow$I: $\mathcal{D} :: \dfrac{\Gamma, x{:}A_1 \vdash e_0 : A_2}{\Gamma \vdash \lambda x.\, e_0 : A_1 \rightarrow A_2}$

    We have either $e' = \lambda x.\, e_0'$ or $e' = ((\lambda x.\, e_0') : As)$, for some $e_0' \sqsupseteq e_0$.

$\Gamma, x{:}A_1 \vdash e_0 : A_2$	Subd.
$\Gamma, x{:}A_1 \vdash e_0'' \downarrow A_2,\ e_0'' \sqsupseteq e_0'$	By IH
$\Gamma \vdash \lambda x.\, e_0'' \downarrow A_1 \to A_2$	By $\to$I
$\lambda x.\, e_0'' \sqsupseteq \lambda x.\, e_0$	By Definition 3.7

If $e' = \lambda x.\, e_0'$, we have shown part (i). Part (ii) follows:

$\Gamma \vdash \lambda x.\, e_0'' \downarrow A_1 \to A_2$	Above
$(\Gamma \vdash A_1 \to A_2) \lesssim (\Gamma \vdash A_1 \to A_2)$	By Corollary 3.2
$\Gamma \vdash (\lambda x.\, e_0'' : (\Gamma \vdash A_1 \to A_2)) \uparrow A_1 \to A_2$	By ctx-anno
$(\lambda x.\, e_0'' : (\Gamma \vdash A_1 \to A_2)) \sqsupseteq \lambda x.\, e_0'$	By $e_0'' \sqsupseteq e_0'$ and Definition 3.7

On the other hand, if $e' = (\lambda x.\, e_0' : As)$ then:

$\Gamma \vdash \lambda x.\, e_0'' \downarrow A_1 \to A_2$	Above
$(\Gamma \vdash A_1 \to A_2) \lesssim (\Gamma \vdash A_1 \to A_2)$	By Corollary 3.2
$\Gamma \vdash (\lambda x.\, e_0'' : (As, \Gamma \vdash A_1 \to A_2)) \uparrow A_1 \to A_2$	By ctx-anno
$(\lambda x.\, e_0'' : (As, \Gamma \vdash A_1 \to A_2)) \sqsupseteq (\lambda x.\, e_0' : As)$	By $e_0'' \sqsupseteq e_0'$ and Definition 3.7

This suffices for part (ii), and to show part (i):

$\Gamma \vdash A_1 \to A_2 \leq A_1 \to A_2$	By reflexivity of $\leq$
$\Gamma \vdash (\lambda x.\, e_0'' : (As, \Gamma \vdash A_1 \to A_2)) \downarrow A_1 \to A_2$	By sub
$(\lambda x.\, e_0'' : (As, \Gamma \vdash A_1 \to A_2)) \sqsupseteq (\lambda x.\, e_0' : As)$	By $e_0'' \sqsupseteq e_0'$ and Definition 3.7

2. **Case** $\to$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e_1 : B \to A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1 e_2 : A}$

   We have $e' = e_1' e_2'$ for some e_1', e_2'.

	$\Gamma \vdash e_1 : B \to A$	Subd.
	$\Gamma \vdash e_1'' \uparrow B \to A,\ e_1'' \sqsupseteq e_1'$	By IH
	$\Gamma \vdash e_2 : B$	Subd.
	$\Gamma \vdash e_2'' \downarrow B,\ e_2'' \sqsupseteq e_2'$	By IH
(ii)☞	$\Gamma \vdash e_1'' e_2'' \uparrow A$	By $\to$E
(i), (ii)☞	$e_1'' e_2'' \sqsupseteq e_1' e_2'$	By Definition 3.7

   We can now show part (i):

	$\Gamma \vdash e_1'' e_2'' \uparrow A$	Above
	$\Gamma \vdash A \leq A$	By reflexivity of subtyping
(i)☞	$\Gamma \vdash e_1'' e_2'' \downarrow A$	By sub

3. **Case** $*\mathrm{E}_1$, $*\mathrm{E}_2$: Essentially simpler instances of the $\to$E case.

4. **Case** $\mathbf{1}$I: $\mathcal{D} :: \dfrac{}{\Gamma \vdash () : \mathbf{1}}$

$e' \sqsupseteq ()$ is given. By the definition of $\sqsupseteq$, either $e' = ()$ or $e' = (() : As)$ for some typing annotations As. In the former case:

(i)☞	$\Gamma \vdash () \downarrow \mathbf{1}$	By **1**I
(i)☞	$() \sqsupseteq ()$	By Definition 3.7
	$(\Gamma \vdash \mathbf{1}) \lesssim (\Gamma \vdash \mathbf{1})$	By Corollary 3.2
(ii)☞	$\Gamma \vdash (() : (\Gamma \vdash \mathbf{1})) \uparrow \mathbf{1}$	By ctx-anno
(ii)☞	$(() : (\Gamma \vdash \mathbf{1})) \sqsupseteq ()$	By Definition 3.7

In the latter case, where $e' = (() : As)$:

	$\Gamma \vdash () \downarrow \mathbf{1}$	By **1**I
(ii)☞	$\Gamma \vdash (() : As, (\Gamma \vdash \mathbf{1})) \uparrow \mathbf{1}$	By ctx-anno
(ii)☞	$(() : As, (\Gamma \vdash \mathbf{1})) \sqsupseteq (() : As)$	By Definition 3.7
(i)☞	$\Gamma \vdash (() : As, (\Gamma \vdash \mathbf{1})) \downarrow \mathbf{1}$	By sub

5. **Case** sub: $\mathcal{D} :: \dfrac{\Gamma \vdash e : B \quad \Gamma \vdash B \leq A}{\Gamma \vdash e : A}$

	$\Gamma \vdash e : B$	Subd.
(i)☞	$\Gamma \vdash e'' \uparrow B,\ e'' \sqsupseteq e'$	By IH
	$\Gamma \vdash B \leq A$	Subd.
(i)☞	$\Gamma \vdash e'' \downarrow A$	By sub
(ii)☞	$(e'' : A) \sqsupseteq e''$	By Definition 3.7
	$(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$	By Corollary 3.2
(ii)☞	$\Gamma \vdash (e'' : (\Gamma \vdash A)) \uparrow A$	By ctx-anno

6. **Case** $*$I: $\mathcal{D} :: \dfrac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash (e_1, e_2) : A_1 * A_2}$

We have $e' \sqsupseteq (e_1, e_2)$. Therefore there exist e'_1, e'_2 such that $e'_1 \sqsupseteq e_1$ and $e'_2 \sqsupseteq e_2$.

$\Gamma \vdash e_1 : A_1$	Subd.
$\Gamma \vdash e''_1 \downarrow A_1,\ e''_1 \sqsupseteq e'_1$	By IH
$\Gamma \vdash e_2 : A_2$	Subd.
$\Gamma \vdash e''_2 \downarrow A_2,\ e''_2 \sqsupseteq e'_2$	By IH
$\Gamma \vdash (e''_1, e''_2) \downarrow A_1 * A_2$	By $*$I
$(e''_1, e''_2) \sqsupseteq (e'_1, e'_2)$	By Definition 3.7

7. **Case** δI: Similar to the $*$I case.

8. **Case** $\wedge$I: $\mathcal{D} :: \dfrac{\Gamma \vdash v : A_1 \quad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}$

$e' \sqsupseteq v$	Given
$\Gamma \vdash v : A_1$	Subd.
$\Gamma \vdash e'' \downarrow A_1,\ e'' \sqsupseteq e'$	By IH
$e'' \sqsupseteq e'$	Above
$e'' \sqsupseteq v$	By Proposition 3.9
$\Gamma \vdash v : A_2$	Subd.
$\Gamma \vdash e''' \downarrow A_2,\ e''' \sqsupseteq e''$	By IH
$\Gamma \vdash e'''' \downarrow A_1,\ e'''' \sqsupseteq_\ell e'''$	By Lemma 3.14
$\Gamma \vdash e'''' \downarrow A_2$	By Lemma 3.13
$e'''' \sqsupseteq v$	By Proposition 3.9
v value	Given
e'''' value	By Lemma 3.10
$\Gamma \vdash e'''' \downarrow A_1 \wedge A_2$	By $\wedge$I

9. **Case** ΠI, $\supset$I, ΠE, ΣI, $\vee I_{1,2}$, $\otimes$I, $\supset$E, $\wedge E_{1,2}$: Similar to the $\wedge$I case, but simpler (for instance, no need to apply monotonicity).

10. **Case** $\vee$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e_0 : B_1 \vee B_2 \quad \begin{array}{l}\Gamma, x{:}B_1 \vdash \mathcal{E}[x] : A \\ \Gamma, y{:}B_2 \vdash \mathcal{E}[y] : A\end{array}}{\Gamma \vdash \mathcal{E}[e_0] : A}\,\vee\mathrm{E}$

By Lemma 3.11, there exist $e_0' \sqsupseteq e_0$ and $\mathcal{E}'$ such that $e' = \mathcal{E}'[e_0']$ and for all e_1, it is the case that $\mathcal{E}'[e_1] \sqsupseteq \mathcal{E}[e_1]$.

$\Gamma \vdash e_0 : B_1 \vee B_2$	Subd.
$e_0' \sqsupseteq e_0$	Given
$\Gamma \vdash e_0'' \uparrow B_1 \vee B_2,\ e_0'' \sqsupseteq e_0'$	By IH
$\Gamma, x{:}B_1 \vdash \mathcal{E}[x] : A$	Subd.
$\mathcal{E}'[x] \sqsupseteq \mathcal{E}[x]$	Above
$\Gamma, x{:}B_1 \vdash e'' \downarrow A,\ e'' \sqsupseteq \mathcal{E}'[x]$	By IH
$\exists \mathcal{E}''.\ e'' = \mathcal{E}''[x],\ \mathcal{E}''[y] \sqsupseteq \mathcal{E}'[y]$	By Lemma 3.11
$\Gamma, y{:}B_2 \vdash \mathcal{E}[y] : A$	Subd.
$\mathcal{E}''[y] \sqsupseteq \mathcal{E}'[y]$	Above
$\Gamma, y{:}B_2 \vdash e''' \downarrow A,\ e''' \sqsupseteq \mathcal{E}''[y]$	By IH
$\exists \mathcal{E}'''.\ e''' = \mathcal{E}'''[y],\ \mathcal{E}'''[x] \sqsupseteq \mathcal{E}''[x]$	By Lemma 3.11
$\Gamma, y{:}B_2 \vdash \mathcal{E}'''[y] \downarrow A$	By $e''' = \mathcal{E}'''[y]$ and $\Gamma, y{:}B_2 \vdash e''' \downarrow A$

$\mathcal{E}'''[x] \sqsupseteq \mathcal{E}''[x]$	Above
$\Gamma, x{:}B_1 \vdash e'''' \downarrow A,\ e'''' \sqsupseteq_\ell \mathcal{E}'''[x]$	By Lemma 3.14
$\exists \mathcal{E}''''.\quad e'''' = \mathcal{E}''''[x],$ $\mathcal{E}''''[y] \sqsupseteq_\ell \mathcal{E}'''[y]$	By Definition 3.8
$\Gamma, y{:}B_2 \vdash \mathcal{E}''''[y] \downarrow A$	By Lemma 3.13
$\Gamma \vdash \mathcal{E}''''[e_0''] \downarrow A$	By $\vee$E
$\mathcal{E}''''[e_0''] \sqsupseteq \mathcal{E}'[e_0']$	By Proposition 3.9 and Definition 3.7

11. **Case** direct, $\perp$E, ΣE, $\otimes$E: Similar to the $\vee$E case, but simpler (for instance, no need to apply monotonicity).

12. **Case** var: $\mathcal{D} :: \dfrac{\Gamma(x) = A}{\Gamma \vdash x : A}$

    We have $e' \sqsupseteq e$. By definition of $\sqsupseteq$, there can be no annotations on variables, and so $e' = x$.

    $\Gamma \vdash x \uparrow A$ By var

13. **Case** fixvar: Similar to the preceding case.

14. **Case** $\top$I: $\mathcal{D} :: \dfrac{\Gamma \vdash v \text{ ok}}{\Gamma \vdash v : \top}$

$e' \sqsupseteq v$	Given
e' value	By Lemma 3.10
$\Gamma \vdash e' \downarrow \top$	By $\top$I

15. **Case** contra: $\mathcal{D} :: \dfrac{\overline{\Gamma} \models \perp \qquad \Gamma \vdash e \text{ ok}}{\Gamma \vdash e : A}$

$\overline{\Gamma} \models \perp$	Subd.
$\Gamma \vdash e' \downarrow A$	By contra

□

Corollary 3.16. *If* $\Gamma \vdash e : A$ *then there exists* $e' \sqsupseteq e$ *such that* $\Gamma \vdash e' \downarrow^{\mathsf{tri}} A$ *and there exists* $e'' \sqsupseteq e$ *such that* $\Gamma \vdash e'' \uparrow^{\mathsf{tri}} A$.

Remark 3.17. Theorem 3.15 engenders a procedure for constructing an annotated term from a derivation in the type assignment system. The annotated term may not be the "best" in the sense of having a minimal (to say nothing of *minimum*) set of annotations. For example, we leave the term unchanged in this checking derivation:

$$\dfrac{}{\vdash () : \mathbf{1}} \quad\Longrightarrow\quad \dfrac{}{\vdash () \downarrow \mathbf{1}}$$

Subtyping	$\Gamma \vdash A \leq B$
Contextual subtyping	$(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$
Constraint satisfaction	$\overline{\Gamma} \models P$
Index expression sorting	$\overline{\Gamma} \vdash i : \gamma$
Data constructor typing	$\Gamma \vdash c : A \to \delta(i)$
Simple tridirectional checking	$\Gamma \vdash e \downarrow^{\mathsf{tri}} A$, or $\downarrow A$
Simple tridirectional synthesis	$\Gamma \vdash e \uparrow^{\mathsf{tri}} A$, or $\uparrow A$
Left tridirectional checking	$\Gamma; \Delta \vdash e \downarrow^{\mathbb{L}} A$
Left tridirectional synthesis	$\Gamma; \Delta \vdash e \uparrow^{\mathbb{L}} A$
Δ appear linearly in e	$\Delta \Vdash e$ ok
—and in evaluation position in e	$\Delta \Vvdash e$ ev/ok

Figure 3.7: Judgment forms appearing in this chapter

But if the input indulges in aimless subsumption, we will add a superfluous annotation:

$$\dfrac{\dfrac{}{\vdash () : \mathbf{1}} \quad \mathbf{1} \leq \mathbf{1}}{\vdash () : \mathbf{1}} \quad \Longrightarrow \quad \dfrac{\dfrac{(\vdash \mathbf{1}) \lesssim (\vdash \mathbf{1}) \quad \overline{\vdash () \downarrow \mathbf{1}}}{\vdash (() : (\vdash \mathbf{1})) \uparrow \mathbf{1}} \quad \mathbf{1} \leq \mathbf{1}}{\vdash (() : (\vdash \mathbf{1})) \downarrow \mathbf{1}}$$

3.5 The left tridirectional system

In the simple tridirectional system, the contextual rules are highly nondeterministic. Not only must we choose which contextual rule to apply, but each rule can be applied repeatedly with the same context $\mathcal{E}$; for direct, which does not even break down the type of e', this repeated application is quite pointless. The system in this section has only one contextual rule and disallows repeated application. Inspired by the sequent calculus formulation of Barbanera et al. [BDCd95], it replaces the contextual rules with one contextual rule direct$\mathbb{L}$, closely corresponding to direct, and several *left rules*, shown in the upper right hand corner of Figure 3.8. In combination, these rules subsume the contextual rules of the simple tridirectional system.

The typing judgments in the left tridirectional system are

$$\Gamma; \Delta \vdash e \uparrow^{\mathbb{L}} A \qquad \Gamma; \Delta \vdash e \downarrow^{\mathbb{L}}$$

where Δ is a *linear context* whose domain is a new syntactic category, the *linear variables* $\bar{\mathbf{x}}, \bar{\mathbf{y}}$ and so forth. These linear variables correspond to the variables introduced in evaluation position in the direct rule, and appear exactly once in the term e, in evaluation position. We consider these linear variables to be values, like ordinary variables.

> **Note.** From this point on, we write $\Gamma \vdash e \downarrow^{\mathsf{tri}} A$ and $\Gamma \vdash e \uparrow^{\mathsf{tri}} A$ for judgments in the simple tridirectional system, to more clearly distinguish them from the left tridirectional system. One can always determine a judgment's system by the number of contexts: if

Rules of the tridirectional system absent in the left tridirectional system:

$$\frac{\Gamma \vdash e' \uparrow^{\mathsf{tri}} A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C}\ \mathsf{direct}$$

$$\frac{\Gamma \vdash e' \uparrow^{\mathsf{tri}} \bot \qquad \Gamma \vdash \mathcal{E}[e']\ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C}\ \bot\mathrm{E}$$

$$\frac{\Gamma \vdash e' \uparrow^{\mathsf{tri}} A \vee B \qquad \begin{array}{l}\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] \downarrow^{\mathsf{tri}} C\end{array}}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C}\ \vee\mathrm{E}$$

$$\frac{\Gamma \vdash e' \uparrow^{\mathsf{tri}} \Sigma a{:}\gamma.\, A \qquad \Gamma, a{:}\gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C}\ \Sigma\mathrm{E}$$

$$\frac{\Gamma \vdash e' \uparrow P \otimes A \qquad \Gamma, P, x{:}A \vdash \mathcal{E}[x] \downarrow C}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\ \otimes\mathrm{E}$$

Rules new or substantially altered in the left tridirectional system:

$$\frac{}{\Gamma; \overline{\mathbf{x}}{:}A \vdash \overline{\mathbf{x}} \uparrow^{\mathbb{L}} A}\ \overline{\mathsf{var}}$$

$$\frac{\begin{array}{l}e' \text{ not a linear var} \\ \Gamma; \Delta_1 \vdash e' \uparrow^{\mathbb{L}} A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{E}[e'] \downarrow^{\mathbb{L}} C}\ \mathsf{direct}\mathbb{L}$$

$$\frac{\Gamma \vdash e\ \mathsf{ok} \qquad \Delta, \overline{\mathbf{x}}{:}\bot \Vdash e\ \mathsf{ok}}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\bot \vdash e \downarrow^{\mathbb{L}} C}\ \bot\mathbb{L}$$

$$\frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vee B \vdash e \downarrow^{\mathbb{L}} C}\ \vee\mathbb{L}$$

$$\frac{\Gamma, a{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\Sigma a{:}\gamma.\, A \vdash e \downarrow^{\mathbb{L}} C}\ \Sigma\mathbb{L}$$

$$\frac{\Gamma, P; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}(P \otimes A) \vdash e \downarrow C}\ \otimes\mathbb{L}$$

$$\frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \downarrow^{\mathbb{L}} C}\ \wedge\mathbb{L}_1 \qquad \frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \downarrow^{\mathbb{L}} C}\ \wedge\mathbb{L}_2$$

$$\frac{\overline{\Gamma} \vdash i : \gamma \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}[i/a]A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\Pi a{:}\gamma.\, A \vdash e \downarrow^{\mathbb{L}} C}\ \Pi\mathbb{L}$$

$$\frac{\overline{\Gamma} \models P \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}P \supset A \vdash e \downarrow^{\mathbb{L}} C}\ \supset\mathbb{L}$$

Figure 3.8: Part of the left tridirectional system, with the part of the simple tridirectional system (upper left corner) from which it substantially differs.

Typing rules of the left tridirectional system identical to the simple tridirectional system, except for the linear contexts Δ:

$\boxed{\Gamma;\Delta \vdash e \downarrow A}$ $\boxed{\Gamma;\Delta \vdash e \uparrow A}$

$$\frac{\Gamma(x) = A}{\Gamma;\cdot \vdash x \uparrow A}\,\mathsf{var} \qquad \frac{\Gamma, x{:}A;\cdot \vdash e \downarrow B}{\Gamma;\cdot \vdash \lambda x.\,e \downarrow A \to B}\,{\to}\mathrm{I} \qquad \frac{\Gamma;\Delta_1 \vdash e_1 \uparrow A \to B \qquad \Gamma;\Delta_2 \vdash e_2 \downarrow A}{\Gamma;\Delta_1,\Delta_2 \vdash e_1 e_2 \uparrow B}\,{\to}\mathrm{E}$$

$$\frac{\Gamma;\Delta \vdash e \uparrow A \qquad \Gamma \vdash A \leq B}{\Gamma;\Delta \vdash e \downarrow B}\,\mathsf{sub} \qquad \frac{\Gamma(u) = A}{\Gamma;\cdot \vdash u \uparrow A}\,\mathsf{fixvar} \qquad \frac{\Gamma, u{:}A;\cdot \vdash e \downarrow A}{\Gamma;\cdot \vdash \mathbf{fix}\,u.\,e \downarrow A}\,\mathsf{fix}$$

$$\frac{}{\Gamma;\cdot \vdash () \downarrow \mathbf{1}}\,\mathbf{1}\mathrm{I}$$

$$\frac{\Gamma;\Delta_1 \vdash e_1 \downarrow A_1 \qquad \Gamma;\Delta_2 \vdash e_2 \downarrow A_2}{\Gamma;\Delta_1,\Delta_2 \vdash (e_1, e_2) \downarrow A_1 * A_2}\,{*}\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash e \uparrow A * B}{\Gamma;\Delta \vdash \mathbf{fst}(e) \uparrow A}\,{*}\mathrm{E}_1 \qquad \frac{\Gamma;\Delta \vdash e \uparrow A * B}{\Gamma;\Delta \vdash \mathbf{snd}(e) \uparrow B}\,{*}\mathrm{E}_2$$

$$\frac{\overline{\Gamma} \vdash c : A \to \delta_2(i) \qquad \Gamma \vdash \delta_2(i) \leq \delta_1(j) \qquad \Gamma;\Delta \vdash e \downarrow A}{\Gamma;\Delta \vdash c(e) \downarrow \delta_1(j)}\,\delta\mathrm{I} \qquad \frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e\ \mathsf{ok} \qquad \Delta \Vdash e\ \mathsf{ok}}{\Gamma;\Delta \vdash e \downarrow A}\,\mathsf{contra}$$

$$\frac{\Gamma;\Delta \vdash e \uparrow \delta(i) \qquad \Gamma;\cdot \vdash ms \downarrow_{\delta(i)} C}{\Gamma;\Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ ms \downarrow C}\,\delta\mathrm{E}$$

$$\frac{\Gamma \vdash v\ \mathsf{ok} \qquad \Delta \Vdash v\ \mathsf{ok}}{\Gamma;\Delta \vdash v \downarrow \top}\,\top\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash v \downarrow A \qquad \Gamma;\Delta \vdash v \downarrow B}{\Gamma;\Delta \vdash v \downarrow A \wedge B}\,\wedge\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash e \uparrow A \wedge B}{\Gamma;\Delta \vdash e \uparrow A}\,\wedge\mathrm{E}_1 \qquad \frac{\Gamma;\Delta \vdash e \uparrow A \wedge B}{\Gamma;\Delta \vdash e \uparrow B}\,\wedge\mathrm{E}_2$$

$$\frac{\Gamma, a{:}\gamma;\Delta \vdash v \downarrow A}{\Gamma;\Delta \vdash v \downarrow \Pi a{:}\gamma.\,A}\,\Pi\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash e \uparrow \Pi a{:}\gamma.\,A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma;\Delta \vdash e \uparrow [i/a]\,A}\,\Pi\mathrm{E} \qquad \frac{\Gamma;\Delta \vdash e \downarrow [i/a]\,A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma;\Delta \vdash e \downarrow \Sigma a{:}\gamma.\,A}\,\Sigma\mathrm{I}$$

$$\frac{\Gamma;\Delta \vdash e \downarrow A}{\Gamma;\Delta \vdash e \downarrow A \vee B}\,\vee\mathrm{I}_1 \qquad \frac{\Gamma;\Delta \vdash e \downarrow B}{\Gamma;\Delta \vdash e \downarrow A \vee B}\,\vee\mathrm{I}_2 \qquad \frac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \qquad \Gamma;\Delta \vdash e \downarrow A}{\Gamma;\Delta \vdash (e : (\Gamma_0 \vdash A_0), As) \uparrow A}\,\mathsf{ctx\text{-}anno}$$

$$\frac{\Gamma, P;\Delta \vdash v \downarrow A}{\Gamma;\Delta \vdash v \downarrow P \supset A}\,{\supset}\mathrm{I} \qquad \frac{\Gamma;\Delta \vdash e \uparrow P \supset A \qquad \overline{\Gamma} \models P}{\Gamma;\Delta \vdash e \uparrow A}\,{\supset}\mathrm{E}$$

$$\frac{\Gamma;\Delta \vdash e \downarrow A \qquad \overline{\Gamma} \models P}{\Gamma;\Delta \vdash e \downarrow P \otimes A}\,{\otimes}\mathrm{I}$$

Figure 3.9: The part of the left tridirectional system substantially similar from the simple tridirectional system. The figure also summarizes the simple tridirectional system: The complete typing rules for the simple tridirectional system can be obtained by removing the second context Δ, including premises of the form $\Delta \Vdash e$ ok. Hence the subscripts $\uparrow^{\mathbb{L}}$, $\downarrow^{\mathbb{L}}$ are elided.

$$\boxed{\Gamma; c : A^{con}; c(x) : \delta(i) \vdash e \downarrow^{\mathbb{L}} C}$$

$$\frac{\delta \preceq \delta' \qquad \Gamma, x{:}A, i \doteq i'; \cdot \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e \downarrow^{\mathbb{L}} C}\ \delta\text{S-ct} \qquad \frac{\delta \npreceq \delta' \qquad \Gamma, x{:}A \vdash e\ \mathsf{ok} \qquad \cdot \Vdash e\ \mathsf{ok}}{\Gamma; c : A \to \delta(i); c(x) : \delta'(i') \vdash e \downarrow^{\mathbb{L}} C}\ \delta\text{F-ct}$$

$$\frac{\begin{array}{c}\Gamma; c : A_1; c(x) : B \vdash e \downarrow^{\mathbb{L}} C \\ \Gamma; c : A_2; c(x) : B \vdash e \downarrow^{\mathbb{L}} C\end{array}}{\Gamma; c : A_1 \wedge A_2; c(x) : B \vdash e \downarrow^{\mathbb{L}} C}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A; c(x) : B \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; c : \Pi a{:}\gamma.\, A; c(x) : B \vdash e \downarrow^{\mathbb{L}} C}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A; c(x) : B \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; c : (P \supset A); c(x) : B \vdash e \downarrow^{\mathbb{L}} C}\ \supset\text{-ct}$$

$$\boxed{\Gamma \vdash ms \downarrow^{\mathbb{L}}{}_{B}\ C}$$

$$\frac{}{\Gamma; \cdot \vdash \cdot \downarrow^{\mathbb{L}}{}_{B}\ C}\ \text{emptyms} \qquad \frac{\Gamma; c : \mathcal{S}(c); c(x) : B \vdash e \downarrow^{\mathbb{L}} C \qquad \Gamma; \cdot \vdash ms \downarrow^{\mathbb{L}}{}_{B}\ C}{\Gamma; \cdot \vdash (c(x) \Rightarrow e \mid ms) \downarrow^{\mathbb{L}}{}_{B}\ C}\ \text{casearm}$$

Figure 3.10: Case typing rules in the left tridirectional system, identical except for added empty linear contexts

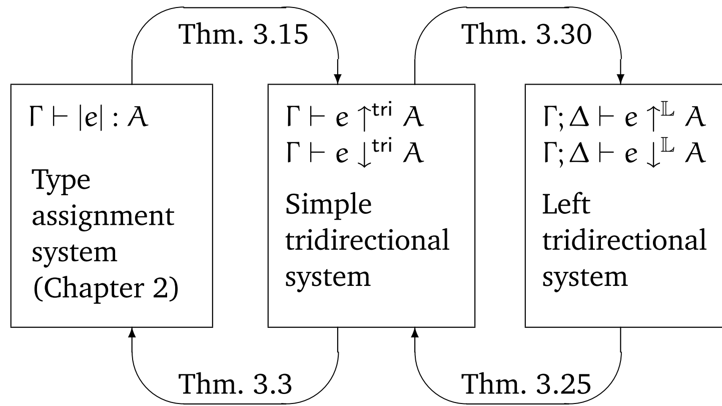


Figure 3.11: Connections between our type systems; see also Figure 5.10

there is one (Γ) it must be in the simple tridirectional system ($\downarrow^{\mathsf{tri}}\uparrow^{\mathsf{tri}}$), if there are two ($\Gamma;\Delta$) it must be in the left tridirectional system ($\downarrow^{\mathbb{L}}\uparrow^{\mathbb{L}}$).

(The next chapter leaves the simple tridirectional system behind and uses unadorned $\downarrow$ and $\uparrow$ symbols for judgments in the *left* tridirectional system.)

The rule direct$\mathbb{L}$ is the only rule that adds to the linear context, and is the true source of linearity: $\overline{\mathbf{x}}$ appears exactly once in evaluation position in $\mathcal{E}[\overline{\mathbf{x}}]$. It requires that the subterm e' being brought out cannot itself be a linear variable, so one cannot bring out a term more than once, unlike with direct.

To maintain linearity, the linear context is split among subterms. For example, in $*$I (Figure 3.8), the context $\Delta = \Delta_1, \Delta_2$ is split between e_1 and e_2. To maintain the property that linear variables appear in evaluation position, the linear context is empty in rules such as $\rightarrow$I, which cannot possibly have a linear variable in evaluation position in its subject.

After some preliminary definitions and lemmas, we prove that this new *left tridirectional system* is sound and complete with respect to the simple tridirectional system from Section 3.3. (See also Figure 3.11).

Definition 3.18. Let $FLV(e)$ denote the set of linear variables appearing free in e. Furthermore, let $\Delta \Vdash e\ \mathsf{ok}$ if and only if (1) for every $\overline{\mathbf{x}} \in \mathrm{dom}(\Delta)$, $\overline{\mathbf{x}}$ appears exactly once in e, and (2) $FLV(e) \subseteq \mathrm{dom}(\Delta)$. (Similarly define $FLV(ms)$ and $\Delta \Vdash ms\ \mathsf{ok}$.)

Proposition 3.19 (Linearity). *If* $\Gamma;\Delta \vdash e \uparrow^{\mathbb{L}} C$ *or* $\Gamma;\Delta \vdash e \downarrow^{\mathbb{L}} C$ *then* $\Delta \Vdash e\ \mathsf{ok}$. *Similarly, if* $\Gamma;\Delta \vdash ms \downarrow_{\delta(i)} C$ *then* $\Delta \Vdash ms\ \mathsf{ok}$.

Proof. By induction on the derivation. For contra, $\top$I, $\bot\mathbb{L}$, δF-ct use the appropriate premise. For all cases in which the term and linear context of each premise are the same as the term and linear context of the conclusion, simply apply the IH. Cases for rules that decompose the term (such as $*\mathrm{E}_1$) require an additional step, such as from $\Delta \Vdash e'\ \mathsf{ok}$ to $\Delta \Vdash \mathbf{fst}(e')\ \mathsf{ok}$; the cases for matches all have empty linear contexts. □

Definition 3.20. Let $\Delta \Vvdash e\ \mathsf{ev/ok}$ if and only if (1) for every $\overline{\mathbf{x}} \in \mathrm{dom}(\Delta)$, there exists an $\mathcal{E}$ such that $e = \mathcal{E}[\overline{\mathbf{x}}]$ and $\overline{\mathbf{x}} \notin FLV(\mathcal{E})$, and (2) $FLV(e) \subseteq \mathrm{dom}(\Delta)$. (It is clear that $\Delta \Vvdash e\ \mathsf{ev/ok}$ implies $\Delta \Vdash e\ \mathsf{ok}$.)

Remark 3.21. Note that the assertion

$$\text{“If } \Gamma;\Delta \vdash e \uparrow^{\mathbb{L}} C \ \text{ or } \ \Gamma;\Delta \vdash e \downarrow^{\mathbb{L}} C \ \text{ then } \ \Delta \Vvdash e\ \mathsf{ev/ok}\text{”}$$

does not hold. Suppose $\Gamma;\overline{\mathbf{x}} \vdash \overline{\mathbf{x}}(e_2e_3) \uparrow^{\mathbb{L}} A \rightarrow B$ and $\Gamma;\overline{\mathbf{y}} \vdash \overline{\mathbf{y}} \downarrow^{\mathbb{L}} A$. It is the case that $\overline{\mathbf{x}} \Vvdash \overline{\mathbf{x}}(e_2e_3)\ \mathsf{ev/ok}$ and $\overline{\mathbf{y}} \Vvdash \overline{\mathbf{y}}\ \mathsf{ev/ok}$. We can apply $\rightarrow$E to get $\Gamma;\overline{\mathbf{x}},\overline{\mathbf{y}} \vdash (\overline{\mathbf{x}}(e_2e_3))\,\overline{\mathbf{y}}$, but $\overline{\mathbf{y}}$ is not in evaluation position in $(\overline{\mathbf{x}}(e_2e_3))\,\overline{\mathbf{y}}$ so we do not have $\overline{\mathbf{x}},\overline{\mathbf{y}} \Vvdash (\overline{\mathbf{x}}(e_2e_3))\,\overline{\mathbf{y}}\ \mathsf{ev/ok}$. However, the following lemma does hold.

Lemma 3.22. *If $\mathcal{D}$ derives* $\Gamma;\Delta \vdash e \uparrow^{\mathbb{L}} C$ *or* $\Gamma;\Delta \vdash e \downarrow^{\mathbb{L}} C$ *by a rule $\mathcal{R}$ and* $\Delta \Vvdash e\ \mathsf{ev/ok}$*, then for each premise* $\Gamma';\Delta' \vdash e' \uparrow^{\mathbb{L}} C'$ *or* $\Gamma';\Delta' \vdash e' \downarrow^{\mathbb{L}} C'$ *of $\mathcal{R}$, it is the case that* $\Delta' \Vvdash e'\ \mathsf{ev/ok}$.

Proof. The proposition is obvious if the linear context is empty in the premise, and for rules where the terms in the premises are identical to the conclusion's term. If the term in each premise is e' where there exists an $\mathcal{E}$ such that the term of the conclusion is $\mathcal{E}[e']$, the linear contexts are identical, and a linear variable appears in evaluation position in e, then it also appears in evaluation position in e'. This takes care of anno, $*E_1$, $*E_2$, and δI. For $\to E$, $*I$, δE, and direct$\mathbb{L}$, we use the fact that if $\mathcal{E}[\overline{\mathbf{x}}] = e$, e' is a subterm of e, and $\overline{\mathbf{x}}$ appears linearly in e and e', there must be some $\mathcal{E}'$ such that $\mathcal{E}'[\overline{\mathbf{x}}] = e'$. □

3.5.1 Soundness

Definition 3.23. A *renaming* ρ is a variable-for-variable substitution from one set of variables ($\mathrm{dom}(\rho)$) to another, disjoint set.

When a renaming is applied to a term, $[\rho]e$, it behaves as a substitution, and can substitute the same variable for multiple variables. Unlike a substitution, however, it can also be applied to contexts. A renaming from linear variables to ordinary program variables, $\rho = x/\overline{\mathbf{x}}, \ldots$, may be applied to a linear context Δ: $[\rho]\Delta$ yields an ordinary context Γ by renaming all variables in $\mathrm{dom}(\Delta)$. In the other direction, a renaming ρ from ordinary program variables to linear variables may be applied to an ordinary context Γ: $[\rho]\Gamma$ yields a zoned context $\Gamma'; \Delta$, where $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) - \mathrm{dom}(\rho)$ and $\mathrm{dom}(\Delta)$ is the image of ρ on Γ restricted to $\mathrm{dom}(\rho)$.

Definition 3.24. $([\rho]\Gamma)_{\Delta}$ denotes Δ where $[\rho]\Gamma = \Gamma'; \Delta$.

Theorem 3.25 (Soundness, Left Tridirectional System)**.** *If* ρ *renames linear variables to ordinary program variables and* $\Gamma; \Delta \vdash e \uparrow^{\mathbb{L}} C$ *(resp.* $\Gamma; \Delta \vdash e \downarrow^{\mathbb{L}} C$*) and* $\Delta \Vdash e$ ev/ok *and* $\mathrm{dom}(\rho) \supseteq \mathrm{dom}(\Delta)$*, then* $\Gamma, [\rho]\Delta \vdash [\rho]e \uparrow^{\mathsf{tri}} C$ *(resp.* $\Gamma, [\rho]\Delta \vdash [\rho]e \downarrow^{\mathsf{tri}} C$*).*

Remark 3.26. The condition $\Delta \Vdash e$ ev/ok is trivially satisfied if $\Delta = \cdot$ and e contains no linear variables, which is precisely the situation for the whole program.

Proof. By induction on the typing derivation. We silently use Lemma 3.22 to satisfy the linearity condition whenever we apply the IH. Most cases are completely straightforward, except for the rules not present in the simple tridirectional system: $\overline{\text{var}}$ and the left rules.

1. **Case** $\overline{\text{var}}$: The derivation is $\Gamma; \overline{\mathbf{x}}{:}A \vdash \overline{\mathbf{x}} \uparrow^{\mathbb{L}} A$. By var,

$$[\rho]\Gamma, [\rho]\overline{\mathbf{x}}{:}A \vdash [\rho]\overline{\mathbf{x}} \uparrow^{\mathsf{tri}} A$$

2. **Case** direct$\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma; \Delta_1 \vdash e' \uparrow^{\mathbb{L}} A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{E}[e'] \downarrow^{\mathbb{L}} C}$

   Let x be new.

$$\begin{array}{rl}
\Gamma, [\rho]\Delta_1 \vdash [\rho]e' \uparrow^{\mathsf{tri}} A & \text{By IH} \\
\Gamma, [\rho]\Delta_1, [\rho]\Delta_2 \vdash [\rho]e' \uparrow^{\mathsf{tri}} A & \text{By weakening}
\end{array}$$

$$\begin{array}{rl}
\Gamma, [\rho, x/\overline{\mathbf{x}}](\Delta_2, \overline{\mathbf{x}}{:}A) \vdash [\rho, x/\overline{\mathbf{x}}]\, \mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathsf{tri}} C & \text{By IH} \\
\Gamma, [\rho]\Delta_2, x{:}A \vdash [\rho]\, \mathcal{E}[x] \downarrow^{\mathsf{tri}} C & \text{By def. of subst. } (\Delta_1, \Delta_2 \Vdash \mathcal{E}[e'] \text{ ev/ok and } \overline{\mathbf{x}} \notin FV(\mathcal{E})) \\
\Gamma, [\rho]\Delta_1, [\rho]\Delta_2, x{:}A \vdash [\rho]\, \mathcal{E}[x] \downarrow^{\mathsf{tri}} C & \text{By weakening}
\end{array}$$

$$\Gamma, [\rho]\Delta_1, [\rho]\Delta_2 \vdash [\rho]\, \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C \quad \text{By direct}$$

3. **Case** $\bot\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma \vdash e \text{ ok} \qquad \Delta, \overline{\mathbf{x}}{:}\bot \Vdash e \text{ ok}}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\bot \vdash e \downarrow^{\mathbb{L}} C}$

Let $x = [\rho]\overline{\mathbf{x}}$.

$$\begin{array}{rl}
\exists \mathcal{E}.\ \ e = \mathcal{E}[\overline{\mathbf{x}}] & \text{By } \Delta, \overline{\mathbf{x}}{:}\bot \Vdash e \text{ ev/ok} \\
\Gamma, [\rho]\Delta, x{:}\bot \vdash x \uparrow^{\mathsf{tri}} \bot & \text{By var} \\
\Gamma, [\rho]\Delta, x{:}\bot \vdash [\rho]\, \mathcal{E}[x] \downarrow^{\mathsf{tri}} \bot & \text{By } \bot\text{E} \\
\Gamma, [\rho]\Delta, x{:}\bot \vdash [\rho]\, e \downarrow^{\mathsf{tri}} \bot & \text{By } e = \mathcal{E}[\overline{\mathbf{x}}] \text{ and def. of subst.}
\end{array}$$

4. **Case** $\vee\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vee B \vdash e \downarrow^{\mathbb{L}} C}$

Let $\rho' = \rho - \overline{\mathbf{x}}$. Let $\rho_A = \rho', x_A/\overline{\mathbf{x}}$, where x_A is new. Let $\rho_B = \rho', x_B/\overline{\mathbf{x}}$, where x_B is new.

$$e = \mathcal{E}[\overline{\mathbf{x}}] \qquad \text{By } \Delta, \overline{\mathbf{x}}{:}A{\vee}B \Vdash e \text{ ev/ok}$$

$$\begin{array}{rl}
\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C & \text{Subderivation} \\
\Gamma, [\rho_A]\Delta, x_A{:}A \vdash [\rho_A]e \downarrow^{\mathsf{tri}} C & \text{By IH} \\
\Gamma, [\rho']\Delta, [\rho_A]\overline{\mathbf{x}}{:}A \vdash [\rho']\ \mathcal{E}[x_A] \downarrow^{\mathsf{tri}} C & \rho_A = \rho', x_A/\overline{\mathbf{x}} \\
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B, x_A{:}A \vdash [\rho']\ \mathcal{E}[x_A] \downarrow^{\mathsf{tri}} C & \text{By weakening} \\
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B, x_A{:}A \vdash ([\rho]\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C & \text{By } \Delta, \overline{\mathbf{x}}{:}A{\vee}B \Vdash \mathcal{E}[\overline{\mathbf{x}}] \text{ ev/ok} \\
\Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C & \text{Subderivation} \\
\Gamma, [\rho_B]\Delta, [\rho_B]\overline{\mathbf{x}}{:}B \vdash [\rho']\ \mathcal{E}[x_B] \downarrow^{\mathsf{tri}} C & \text{By IH} \\
\Gamma, [\rho']\Delta, x_B{:}B \vdash [\rho']\ \mathcal{E}[x_B] \downarrow^{\mathsf{tri}} C & \rho_B = \rho', x_B/\overline{\mathbf{x}} \\
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B, x_B{:}B \vdash [\rho']\ \mathcal{E}[x_B] \downarrow^{\mathsf{tri}} C & \text{By weakening} \\
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B, x_B{:}B \vdash ([\rho]\ \mathcal{E})[x_B] \downarrow^{\mathsf{tri}} C & \text{By } \Delta, \overline{\mathbf{x}}{:}A{\vee}B \Vdash \mathcal{E}[\overline{\mathbf{x}}] \text{ ev/ok}
\end{array}$$

$$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B \vdash [\rho]\overline{\mathbf{x}} \uparrow^{\mathsf{tri}} A \vee B \qquad \text{By var}$$

$$\begin{array}{rl}
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B \vdash ([\rho]\ \mathcal{E})[x] \downarrow^{\mathsf{tri}} C & \text{By } \vee\text{E} \\
\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\vee}B \vdash [\rho]\ \mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathsf{tri}} C & \text{By } e = \mathcal{E}[\overline{\mathbf{x}}] \text{ and subst.}
\end{array}$$

5. **Case** $\Sigma\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma, \alpha{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A \vdash e \downarrow^{\mathbb{L}} C}$

   Let $\rho' = \rho - \overline{\mathbf{x}}$. Let $\rho_A = \rho', x_A/\overline{\mathbf{x}}$, where x_A is new.

$e = \mathcal{E}[\overline{\mathbf{x}}]$	By $\Delta, \overline{\mathbf{x}}{:}A \Vdash e$ ev/ok
$\Gamma, \alpha{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C$	Subderivation
$\Gamma, \alpha{:}\gamma, [\rho_A]\Delta, [\rho_A]\overline{\mathbf{x}}{:}A \vdash [\rho_A]e \downarrow^{\mathsf{tri}} C$	By IH
$\Gamma, \alpha{:}\gamma, [\rho']\Delta, x_A{:}A \vdash ([\rho']\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	$\rho_A = \rho', x_A/\overline{\mathbf{x}}$ and x_A is new
$\Gamma, \alpha{:}\gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A, x_A{:}A \vdash ([\rho']\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	By weakening
$\Gamma, \alpha{:}\gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A, x_A{:}A \vdash ([\rho_A]\mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	By defn. of subst.
$\Gamma, \alpha{:}\gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A, x_A{:}A \vdash ([\rho]\mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	By $\Delta, \overline{\mathbf{x}}{:}A \Vdash \mathcal{E}[\overline{\mathbf{x}}]$ ev/ok
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A \vdash [\rho]\overline{\mathbf{x}} \uparrow^{\mathsf{tri}} \Sigma\alpha{:}\gamma.\, A$	By var
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A \vdash ([\rho]\mathcal{E})[[\rho]\overline{\mathbf{x}}] \downarrow^{\mathsf{tri}} C$	By ΣE
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}\Sigma\alpha{:}\gamma.\, A \vdash [\rho]e \downarrow^{\mathsf{tri}} C$	By $e = \mathcal{E}[\overline{\mathbf{x}}]$ and subst.

6. **Case** $\otimes\mathbb{L}$: Similar to the $\Sigma\mathbb{L}$ case.

7. **Case** $\wedge\mathbb{L}_1$: $\mathcal{D} :: \dfrac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \downarrow^{\mathbb{L}} C}$

   Let $\rho' = \rho - \overline{\mathbf{x}}$. Let $\rho_A = \rho', x_A/\overline{\mathbf{x}}$, where x_A is new.

$e = \mathcal{E}[\overline{\mathbf{x}}]$	By $\Delta, \overline{\mathbf{x}}{:}A \wedge B \Vdash e$ ev/ok
$\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C$	Subderivation
$\Gamma, [\rho_A]\Delta, [\rho_A]\overline{\mathbf{x}}{:}A \vdash [\rho_A]e \downarrow^{\mathsf{tri}} C$	By IH
$\Gamma, [\rho']\Delta, x_A{:}A \vdash ([\rho']\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	$\rho_A = \rho', x_A/\overline{\mathbf{x}}$ and x_A is new
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B, x_A{:}A \vdash ([\rho']\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	By weakening
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B, x_A{:}A \vdash ([\rho]\ \mathcal{E})[x_A] \downarrow^{\mathsf{tri}} C$	By $\Delta, \overline{\mathbf{x}}{:}A{\wedge}B \Vdash \mathcal{E}[\overline{\mathbf{x}}]$ ev/ok
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B \vdash [\rho]\overline{\mathbf{x}} \uparrow^{\mathsf{tri}} A \wedge B$	By var
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B \vdash [\rho]\overline{\mathbf{x}} \uparrow^{\mathsf{tri}} A$	By $\wedge E_1$
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B \vdash ([\rho]\ \mathcal{E})[[\rho]\overline{\mathbf{x}}] \downarrow^{\mathsf{tri}} C$	By direct
$\Gamma, [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B \vdash [\rho]e \downarrow^{\mathsf{tri}} C$	By $e = \mathcal{E}[\overline{\mathbf{x}}]$ and subst.
$\Gamma, [\rho](\Delta, \overline{\mathbf{x}}{:}A{\wedge}B) \vdash [\rho]e \downarrow^{\mathsf{tri}} C$	By $[\rho](\Delta, \overline{\mathbf{x}}{:}A{\wedge}B) = [\rho']\Delta, [\rho]\overline{\mathbf{x}}{:}A{\wedge}B$

Cases $\wedge\mathbb{L}_2$, $\Pi\mathbb{L}$, and $\supset\mathbb{L}$ are analogous to the previous case. □

3.5.2 Completeness

We now show completeness: If a term can be typed in the simple tridirectional system, it can be typed in the left tridirectional system. First, we prove a strengthening lemma and a lemma about

linear variables.

Lemma 3.27 (Strengthening)**.** *If* $\Gamma, y{:}A, \Gamma' \vdash e \underline{\downarrow^{tri}\uparrow^{tri}}\ C$ *where* $y \notin FV(e)$ *then* $\Gamma, \Gamma' \vdash e \underline{\downarrow^{tri}\uparrow^{tri}}\ C$.

Proof. By induction on the given derivation. In most cases, including $\bot$E, simply apply the IH to all premises (note that if y is not free in e it is not free in any subterm of e—as always, we rename bound variables where needed to avoid capture) and apply the same rule.

In the var case we have $e = x$; it is given that $y \notin FV(e)$ so $y \neq x$. We have $(\Gamma, y{:}A, \Gamma')(x) = C$, but $y \neq x$ so also $(\Gamma, \Gamma')(x) = C$. By rule var, $\Gamma, \Gamma' \vdash x \uparrow^{tri} C$.

In the contra case, use $\overline{\Gamma, y{:}A, \Gamma'} = \overline{\Gamma, \Gamma'}$. □

Remark 3.28. An analogous strengthening lemma appears to hold for the left tridirectional system (even, rather vacuously, for linear variables: if $\Gamma; \Delta \vdash e \downarrow C$ then $\Delta \Vdash e$ ok, so the condition that the variable to be removed is not free in e can never be satisfied), but it is not required.

Lemma 3.29. *If* $\Gamma; \bar{\mathbf{x}}{:}A \vdash \bar{\mathbf{x}} \uparrow^{\mathbb{L}} B$ *and* $\Gamma; \Delta, \bar{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C$ *then* $\Gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C$.

Proof. By induction on the first derivation. The only rules that could have been used are $\overline{\text{var}}$, $\wedge E_{1,2}$, ΠE, and $\supset$E. In the $\overline{\text{var}}$ case, $A = B$ so the result is immediate. We show the $\wedge E_1$ case.

- **Case** $\wedge E_1$: $\mathcal{D} :: \dfrac{\Gamma; \bar{\mathbf{x}}{:}A \vdash \bar{\mathbf{x}} \uparrow^{\mathbb{L}} B \wedge B'}{\Gamma; \bar{\mathbf{x}}{:}A \vdash \bar{\mathbf{x}} \uparrow^{\mathbb{L}} B}$

$\Gamma; \Delta, \bar{\mathbf{x}}{:}B \vdash e \downarrow^{\mathbb{L}} C$	Given
$\Gamma; \Delta, \bar{\mathbf{x}}{:}B \wedge B' \vdash e \downarrow^{\mathbb{L}} C$	By $\wedge\mathbb{L}_1$
$\Gamma; \bar{\mathbf{x}}{:}A \vdash \bar{\mathbf{x}} \uparrow^{\mathbb{L}} B \wedge B'$	Subd.
$\Gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow^{\mathbb{L}} C$	By IH

□

Theorem 3.30 (Completeness, Left Rule System)**.** *If* ρ *is a renaming from ordinary program variables to linear variables and* $\Gamma \vdash e \uparrow^{tri} C$ *(resp.* $\Gamma \vdash e \downarrow^{tri} C$*) and* $([\rho]\Gamma)_{\Delta} \Vdash [\rho]e$ ev/ok*, then* $[\rho]\Gamma \vdash [\rho]e \uparrow^{\mathbb{L}} C$ *(resp.* $[\rho]\Gamma \vdash [\rho]e \downarrow^{\mathbb{L}} C$*).*

Proof. By induction on the typing derivation. Most cases are straightforward, and the $\rightarrow$E case is given as a representative example. The more interesting cases are those where the rule used does not exist in the left rule system.

1. **Case** direct: $\mathcal{D} :: \dfrac{\Gamma \vdash e' \uparrow^{tri} A \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{tri} C}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{tri} C}$

   Suppose $[\rho]e' \neq \bar{\mathbf{y}}$ for any $\bar{\mathbf{y}}$. In this case, we can apply direct$\mathbb{L}$, so the proof is easy. Let ρ' be ρ restricted to variables appearing in e', and let $\rho_0 = (\rho - \rho'), \bar{\mathbf{x}}/x$ where $\bar{\mathbf{x}}$ is new. We have $([\rho]\Gamma)_{\Delta} \Vdash [\rho]\,\mathcal{E}[e']$ ev/ok.

$\Gamma \vdash e' \uparrow^{\mathsf{tri}} A$	Subderivation
$[\rho']\Gamma \vdash [\rho']e' \uparrow^{\mathbb{L}} A$	By IH with subst. ρ'
$[\rho]\Gamma \vdash [\rho]e' \uparrow^{\mathbb{L}} A$	By defn. of subst.
$\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C$	Subderivation
$[\rho_0]\,(\Gamma, x{:}A) \vdash [\rho_0]\,\mathcal{E}[x] \downarrow^{\mathbb{L}} C$	By IH with subst. ρ_0
$[\rho]\Gamma, \overline{\mathbf{x}}{:}A \vdash [\rho]\,\mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	By defn. of ρ_0
$[\rho]\Gamma \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$	By direct$\mathbb{L}$

On the other hand, suppose $[\rho]e' = \overline{\mathbf{y}}$ for some $\overline{\mathbf{y}}$. Then e' must be a variable; suppose $\Gamma(e') = B$. Let Γ' be Γ with $e'{:}B$ omitted.

$\Gamma \vdash e' \uparrow^{\mathsf{tri}} A$	Subderivation
$[\rho]\Gamma', \overline{\mathbf{y}}{:}B \vdash \overline{\mathbf{y}} \uparrow^{\mathbb{L}} A$	By IH
$\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C$	Subderivation
$e' \notin FV(\mathcal{E}[x])$	By $([\rho]\Gamma)_{\Delta} \Vdash [\rho]\,\mathcal{E}[e']$ ok
$\Gamma', x{:}A \vdash \mathcal{E}[x] \downarrow^{\mathsf{tri}} C$	By Lemma 3.27
$[\rho, \overline{\mathbf{y}}/x](\Gamma', x{:}A) \vdash [\rho, \overline{\mathbf{y}}/x]\,\mathcal{E}[x] \downarrow^{\mathbb{L}} C$	By IH
$[\rho]\Gamma', \overline{\mathbf{y}}{:}A \vdash [\rho]\,\mathcal{E}[\overline{\mathbf{y}}] \downarrow^{\mathbb{L}} C$	Applying substs.
$[\rho]\Gamma', \overline{\mathbf{y}}{:}B \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$	By Lemma 3.29

2. **Case** $\bot$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e' \uparrow^{\mathsf{tri}} \bot \qquad \Gamma \vdash \mathcal{E}[e']\ \mathsf{ok}}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\mathsf{tri}} C}$

Suppose $[\rho]e' \neq \overline{\mathbf{y}}$ for any $\overline{\mathbf{y}}$. Let $\overline{\mathbf{x}}$ be new. Let ρ_1 be ρ restricted to variables appearing in e'. Let $\rho_2 = \rho - \rho_1$.

$\Gamma \vdash e' \uparrow^{\mathsf{tri}} \bot$	Subderivation
$[\rho_1]\Gamma \vdash [\rho_1]e' \uparrow^{\mathbb{L}} \bot$	By IH
$([\rho]\Gamma)_{\Delta}, \overline{\mathbf{x}}{:}\bot \Vdash [\rho]\,(\mathcal{E}[e'])$ ok	Given
$([\rho_2]\Gamma)_{\Delta}, \overline{\mathbf{x}}{:}\bot \Vdash ([\rho_2]\,\mathcal{E})[\overline{\mathbf{x}}]$ ok	
$[\rho_2]\Gamma, \overline{\mathbf{x}}{:}\bot \vdash ([\rho_2]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	By $\bot\mathbb{L}$
$[\rho]\Gamma \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$	By direct$\mathbb{L}$

Otherwise, suppose $[\rho]e' = \overline{\mathbf{x}}$ for some $\overline{\mathbf{x}}$. Then e' must be a variable. Suppose $\Gamma(e') = A$. Let Γ' be Γ without $e'{:}A$.

$([\rho]\Gamma)_{\Delta} \Vdash [\rho]\,\mathcal{E}[e']$ ok	Given
$([\rho]\Gamma')_{\Delta}, \overline{\mathbf{x}}{:}\bot \Vdash ([\rho]\,\mathcal{E})[[\rho]e']$ ok	Γ' is Γ without $e'{:}A$
$([\rho]\Gamma')_{\Delta}, \overline{\mathbf{x}}{:}\bot \Vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}]$ ok	By $[\rho]e' = \overline{\mathbf{x}}$
$[\rho]\Gamma', \overline{\mathbf{x}}{:}\bot \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	By $\bot\mathbb{L}$

$\Gamma \vdash e' \uparrow^{\text{tri}} \bot$	Subderivation
$([\overline{\mathbf{x}}/e']\Gamma)_{\Delta} \vdash \overline{\mathbf{x}} \uparrow^{\mathbb{L}} \bot$	By IH
$[\rho]\Gamma', \overline{\mathbf{x}}{:}A \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	By Lemma 3.29
$[\rho]\Gamma', \overline{\mathbf{x}}{:}A \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$	By $[\rho]e' = \overline{\mathbf{x}}$ and def. of subst.

3. **Case** $\vee$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e' \uparrow^{\text{tri}} A \vee B \qquad \begin{array}{c}\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\text{tri}} C \\ \Gamma, y{:}B \vdash \mathcal{E}[y] \downarrow^{\text{tri}} C\end{array}}{\Gamma \vdash \mathcal{E}[e'] \downarrow^{\text{tri}} C} \vee\text{E}$

Suppose $[\rho]e' \neq \overline{\mathbf{y}}$ for any $\overline{\mathbf{y}}$. Let ρ' be ρ restricted to variables appearing in e'. Let $\rho_0 = \rho - \rho'$. Let $\rho_x = \rho_0, \overline{\mathbf{x}}/x$ and $\rho_y = \rho_0, \overline{\mathbf{x}}/y$ where $\overline{\mathbf{x}}$ is new.

$\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\text{tri}} C$	Subderivation
$[\rho_x](\Gamma, x{:}A) \vdash [\rho_x]\,\mathcal{E}[x] \downarrow^{\text{tri}} C$	By IH
$[\rho_0]\Gamma, \overline{\mathbf{x}}{:}A \vdash ([\rho_0]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By $\rho_x = \rho_0, \overline{\mathbf{x}}/x$
$\Gamma, y{:}B \vdash \mathcal{E}[y] \downarrow^{\text{tri}} C$	Subderivation
$[\rho_y](\Gamma, y{:}B) \vdash [\rho_y]\,\mathcal{E}[y] \downarrow^{\text{tri}} C$	By IH
$[\rho_0]\Gamma, \overline{\mathbf{x}}{:}B \vdash ([\rho_0]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By $\rho_y = \rho_0, \overline{\mathbf{x}}/y$
$[\rho_0]\Gamma, \overline{\mathbf{x}}{:}A{\vee}B \vdash ([\rho_0]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By $\vee\mathbb{L}$
$[\rho]\Gamma, \overline{\mathbf{x}}{:}A{\vee}B \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By defn. of renaming
$\Gamma \vdash e' \uparrow^{\text{tri}} A \vee B$	Subderivation
$[\rho']\Gamma \vdash [\rho']e' \uparrow^{\mathbb{L}} A \vee B$	By IH
$[\rho]\Gamma \vdash [\rho]e' \uparrow^{\mathbb{L}} A \vee B$	By defn. of renaming
$[\rho]\Gamma \vdash ([\rho]\,\mathcal{E})[[\rho]e'] \downarrow^{\mathbb{L}} C$	By direct$\mathbb{L}$
$[\rho]\Gamma \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$	By def. of subst.

Otherwise, suppose $[\rho]e' = \overline{\mathbf{x}}$ for some $\overline{\mathbf{x}}$. Then e' must be a variable. Suppose $\Gamma(e') = D$. Let Γ' be Γ without $e'{:}D$. Let $\rho_x = \rho, \overline{\mathbf{x}}/x$ and $\rho_y = \rho, \overline{\mathbf{x}}/y$ where $\overline{\mathbf{x}}$ is new.

$\Gamma, x{:}A \vdash \mathcal{E}[x] \downarrow^{\text{tri}} C$	Subderivation
$[\rho_x]\Gamma', \overline{\mathbf{x}}{:}A \vdash [\rho_x]\,\mathcal{E}[x] \downarrow^{\text{tri}} C$	By IH
$[\rho]\Gamma', \overline{\mathbf{x}}{:}A \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By defn. of renaming
$\Gamma, y{:}B \vdash \mathcal{E}[y] \downarrow^{\text{tri}} C$	Subderivation
$[\rho_y]\Gamma', \overline{\mathbf{x}}{:}B \vdash [\rho_y]\,\mathcal{E}[y] \downarrow^{\text{tri}} C$	By IH
$[\rho]\Gamma', \overline{\mathbf{x}}{:}B \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\text{tri}} C$	By defn. of renaming
$[\rho]\Gamma', \overline{\mathbf{x}}{:}A{\vee}B \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	By $\vee\mathbb{L}$
$\Gamma \vdash e' \uparrow^{\text{tri}} A \vee B$	Subderivation
$([\overline{\mathbf{x}}/e']\Gamma)_{\Delta} \vdash \overline{\mathbf{x}} \uparrow^{\mathbb{L}} A \vee B$	By IH

$[\rho]\Gamma', \overline{\mathbf{x}}{:}A \vdash ([\rho]\,\mathcal{E})[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$ By Lemma 3.29
$[\rho]\Gamma', \overline{\mathbf{x}}{:}A \vdash [\rho]\,\mathcal{E}[e'] \downarrow^{\mathbb{L}} C$ By defn. of renaming

4. **Case** ΣE: Similar to the $\vee$E case.

5. **Case** $\otimes$E: Similar to the $\vee$E case.

6. **Case** $\to$E: $\mathcal{D} :: \dfrac{\Gamma \vdash e_1 \uparrow^{\mathsf{tri}} A \to C \qquad \Gamma \vdash e_2 \downarrow^{\mathsf{tri}} A}{\Gamma \vdash e_1 e_2 \uparrow^{\mathsf{tri}} C}$

   Let ρ_1 and ρ_2 be ρ restricted to variables appearing in e_1 and e_2 respectively.

$([\rho]\Gamma)_{\Delta} \Vdash e_1 e_2$	Given
$([\rho_1]\Gamma)_{\Delta} \Vdash e_1$	By defn. of evaluation contexts
$([\rho_2]\Gamma)_{\Delta} \Vdash e_2$	By defn. of evaluation contexts
$\Gamma \vdash e_1 \uparrow^{\mathsf{tri}} A \to C$	Subderivation
$[\rho_1]\Gamma \vdash [\rho_1]e_1 \uparrow^{\mathbb{L}} A \to C$	By IH
$\Gamma \vdash e_2 \downarrow^{\mathsf{tri}} A$	Subderivation
$[\rho_2]\Gamma \vdash [\rho_2]e_2 \downarrow^{\mathbb{L}} A$	By IH
$[\rho]\Gamma \vdash [\rho]\; e_1 e_2 \uparrow^{\mathbb{L}} C$	By $\to$E

□

3.5.3 Decidability of typing

Theorem 3.31. $\Gamma;\Delta \vdash e \downarrow^{\mathbb{L}} A$ *is decidable.*

Proof. We impose an order $<$ on two judgments $\mathcal{J}_1, \mathcal{J}_2$. Each of these may be a checking judgment $\Gamma_k;\Delta_k \vdash e_k \downarrow A_k$, a synthesis judgment $\Gamma_k;\Delta_k \vdash e_k \uparrow A_k$, a match judgment $\Gamma_k;\cdot \vdash ms_k \downarrow_{B_k} A_k$ or a case typing judgment $\Gamma; c : C_k; c(x) : \delta_k(i_k) \vdash e_k \downarrow A_k$.

When ordering terms, we consider linear variables to be smaller than any other terms; for example, $(\overline{\mathbf{x}}, e_2)$ is smaller than (y, e_2)). When ordering types (that is, type expressions), we consider all index expressions to be of equal size.

The ordering on judgments is defined as follows.

1. If e_1 is smaller than e_2 (or ms_1 is smaller than ms_2, etc.) then $\mathcal{J}_1 < \mathcal{J}_2$. If the subject terms/matches are the same size:

2. If one judgment is synthesis and the other is checking, the synthesis judgment is smaller.

3. If one judgment is checking and the other is case typing, the checking judgment is smaller.

4. If both judgments are checking judgments and A_1 is smaller than A_2, then $\mathcal{J}_1 < \mathcal{J}_2$. If both judgments are synthesis judgments, $\Gamma_1 = \Gamma_2$, $\Delta_1 = \Delta_2$, A_1 is at least as small as some type in $\Gamma_1;\Delta_1$ and A_1 is *larger* than A_2, then $\mathcal{J}_1 < \mathcal{J}_2$. Otherwise:

5. If both judgments are case typing judgments and C_1 is smaller than C_2, then $\mathcal{J}_1 < \mathcal{J}_2$.

6. If the number of times any of the type constructors $\vee$, Σ, $\bot$, $\otimes$, $\wedge$, Π, $\top$, $\supset$ appear in Δ_1 is less than the number of times they appear in Δ_2 then $\mathcal{J}_1 < \mathcal{J}_2$.

Now we show that for every rule, each premise is smaller than the conclusion. It is easy to see that in each rule concluding a match judgment or case typing judgment, the premises are smaller than the conclusion. Only rules δE and δS-ct cross from/to checking judgments; in δE the subject ms is smaller than **case** e' **of** ms, and in δS-ct the premise is considered smaller because it is a checking judgment and the subject e is the same.

Turning to the typing rules, for most premises, the first criterion alone makes the premise smaller. The second criterion is for sub. The fourth criterion is needed for rules such as ΠI and ΠE. Note that a synthesis judgment whose type expression becomes larger is considered smaller! Synthesis judgments eventually "bottom out" at rules like ctx-anno and $*\mathrm{E}_1$, in which the term becomes smaller, or at rules var, fixvar or $\overline{\text{var}}$, where the type synthesized is taken from Γ or Δ. Since all the type expressions in Γ and Δ are finite, there is no problem. The sixth criterion is for the left rules, where the term, direction, and type do not change.

The second premise of direct$\mathbb{L}$ is smaller than its conclusion because we consider linear variables to be the smallest terms and direct$\mathbb{L}$ does not permit e' to be a linear variable. □

3.5.4 Type Safety

If $\cdot;\cdot \vdash e \downarrow^{\mathbb{L}} A$ in the left tridirectional system, from Theorem 3.25 we know $\cdot \vdash e \downarrow^{\text{tri}} A$. Then by Theorem 3.3, $\cdot \vdash |e| : A$ in our type assignment system (Chapter 2). That is, erasing type annotations leads to a typing derivation in the type assignment system. It follows from Theorem 2.21, Type Preservation and Progress, that $|e|$ either diverges or evaluates to a value of type A.

3.6 Related work

3.6.1 Refinements, intersections, unions

Index refinements were proposed by Xi and Pfenning [XP99]. As mentioned earlier, the necessary existential quantifier Σ led to difficulties [Xi98] because elaboration must determine the scope of Σ, which is not syntactically apparent in the source program. Xi addressed this by translating programs into a let-normal form before checking index refinements, which is akin to typechecking the original term in evaluation order. Because of the specific form of Xi's translation, our tridirectional system admits more programs, even when restricted to just index refinements and quantifiers. In Chapter 5 we show that a variant of Xi's idea of evaluation-order traversal is applicable in our significantly more complex setting to eliminate the nondeterminism inherent in the (direct$\mathbb{L}$) rule.

3.6.2 Partial inference systems

Our system shares several properties with Pierce and Turner's *local type inference* [PT98]. Their language has subtyping and impredicative polymorphism, making full type inference undecidable. Their partial inference strategy is formulated as a bidirectional system with synthesis and checking judgments, in a style not too far removed from ours. In order to handle parametric polymorphism

without using nonlocal methods such as unification, they infer type arguments to polymorphic functions, which substantially complicates matters compared to our system, which does not have parametric polymorphism (see Section 8.1.1). Hosoya and Pierce [HP99] further discuss this style, particularly its effectiveness in achieving a reasonable number of annotations.

3.6.3 Principal typings

A *principal type* of e is a type that represents all types of e—in some particular context Γ. A *principal typing* [Jim95] of e is a pair (Γ, A) of a context and a type, such that (Γ, A) represents all pairs (Γ', A') such that $\Gamma' \vdash e : A'$. These definitions depend on some idea of representation, which varies from type system to type system, making comparisons between systems difficult; Wells introduced a general notion of representation [Wel02]. Since full type inference seems in any case unattainable, we have not investigated whether principal typings exist for our language. However, the idea of assigning a *typing* (rather than just a type) to a term appears in our system in the form of contextual typing annotations, enabling us to solve some otherwise very unpleasant problems regarding the scope of quantified index variables.

3.7 Conclusion

Chapter 2 developed a type assignment system with a rich set of property type constructors. That system is sound in a standard call-by-value semantics, but is inherently undecidable. Now, by taking a tridirectional version of the type assignment system, we have obtained a rich yet decidable type system. Every program well-typed under the type assignment system has an annotation with *contextual typings* that checks under the tridirectional rules. Contextual typing annotations may be useful in other settings, such as systems of parametric polymorphism in which subtyping is decidable.

In order to show decidability, and as the first important step towards a practical implementation, we also presented a less nondeterministic *left tridirectional system* and proved it to be decidable and sound and complete with respect to the tridirectional system.

Chapter 5 will drastically reduce the nondeterminism in rule $\mathsf{direct}\mathbb{L}$ by forcing the typechecker to (almost always) traverse subterms in evaluation order, while being sound and complete with respect to the left tridirectional system.

Chapter 4

Pattern matching

4.1 Introduction

Our atomic refinement mechanisms of datasorts and indices refine algebraic datatypes, for which **case** is the elimination form. In earlier chapters we allowed only the simplest form of pattern matching: if $e : \delta(i)$ and δ refines a datatype with constructors $c_1, \ldots, c_n$, the only legal form of **case** was

$$\textbf{case}\ e\ \textbf{of}\ c_1(x_1) \Rightarrow e_1 \mid \ldots \mid c_n(x_n) \Rightarrow e_n$$

However, even in small programs, tuple patterns (p_1, p_2), nested patterns $c(p)$, and named patterns x as p are essential to keep **case** expressions reasonably concise. In addition, we may prefer to omit from the program text case arms that are impossible; in Standard ML this leads to a warning that the **case** is nonexhaustive, but type refinements enable the typechecker to deduce the impossibility of many such case arms.[1]

This chapter extends the formal system to include all the interesting pattern forms found in Standard ML. In our setting, this extension is not trivial: precise analysis of named patterns requires two-way information flow as illustrated below. (This example is taken from Section 7.2.4.)

$$\begin{array}{l}\textbf{case}\ \textit{ins1}\ \mathtt{dict}\ \textbf{of} \\ \quad \mathsf{Red}\ (\mathtt{t}\ \mathsf{as}\ (__, \mathsf{Red}\ __, __)) \Rightarrow \mathsf{Black}\ \mathtt{t} \\ \mid \mathsf{Red}\ (\mathtt{t}\ \mathsf{as}\ (__, __, \mathsf{Red}\ __)) \Rightarrow \mathsf{Black}\ \mathtt{t} \\ \mid \mathtt{dict} \Rightarrow \mathtt{dict}\end{array}$$

Here, *ins1* $\mathtt{dict} \uparrow \mathsf{badRoot}$, and the whole **case** is being checked against rbt. When the last case arm is reached, the remaining pattern space is the union of patterns $\mathsf{Empty} \sqcup \mathsf{Black}(__) \sqcup p$ (where p is irrelevant here). Each component of this union is considered, giving us (first) the pattern space Empty and the given pattern $\mathtt{dict}$. The intersection $\mathtt{dict} \sqcap \mathsf{Empty}$ of these, is $(\mathtt{dict}\ \mathsf{as}\ \mathsf{Empty})$. We start out knowing $\mathtt{dict} : \mathsf{badRoot}$ (since that is the type of the scrutinee *ins1* $\mathtt{dict}$); however, upon descending into the pattern $\mathtt{dict}$ as Empty we find that $\mathtt{dict}$ is really Empty, and from $\mathcal{S}(\mathsf{Empty})$ we

[1] This effect can be achieved within the formalism of the earlier chapters by writing a case arm with an obviously ill-typed body, so that the case expression typechecks only if the case arm is skipped entirely.

can deduce that we actually have something black. Since black $\preceq$ rbt $\preceq$ badRoot, proceeding with the assumption $\mathtt{dict}$: black allows us to check $\mathtt{dict} \downarrow$ rbt.

The point is that in the pattern $\mathtt{dict}$ as Empty—more generally, x as p—examining p can provide a more precise type for x. If the pattern checking system ignored this new information, we would be unable to derive $\mathtt{dict} \downarrow$ rbt, since we would only know $\mathtt{dict}$: badRoot, and badRoot $\not\preceq$ rbt. Moreover, the user would have to stultify the case expression by expanding ... | $\mathtt{dict} \Rightarrow \mathtt{dict}$ into ... | Empty $\Rightarrow$ Empty | Black $x \Rightarrow$ Black x, which is clearly unacceptable for patterns of any complexity.

We begin with syntactic foundations: the grammar of patterns, well-formedness, pattern matching, and subtraction and intersection of patterns. Then we extend the operational semantics to support sequential pattern matching (not needed before because we assumed each arm in a **case** was guarded by a distinct constructor). We then give an overview of pattern checking (Section 4.3) and explain the new pattern typing rules, constructor typing rules, and generalized **case** rules. After defining a type assignment version of the system thus constructed, we prove several lemmas and a soundness result (Theorem 4.11) for pattern typing rules, which allows us to prove type safety. Finally, we discuss the implementation of pattern typechecking (Section 4.8) and examine related work in Section 4.9.

4.2 Foundations of pattern checking

4.2.1 Pattern language

The language of patterns (Figure 4.3) comprises pattern variables x (syntactically the same as program variables), the unit pattern $()$, pair patterns (p_1, p_2), the wildcard pattern __, layered patterns x as p, constructor patterns, the empty pattern $\{\}$, and or-patterns $p_1 \sqcup p_2$. The empty pattern is of no use to the programmer and need not appear in the source language, but it lets us write the result of pattern subtraction: $c(x) - c(y) = \{\}$. We likewise include or-patterns with no source representation: for instance, given a datatype with constructors Nil and Cons, the result of $__ - \mathsf{Cons}(__, \mathsf{Nil}(__))$ is $\mathsf{Nil}(__) \sqcup \mathsf{Cons}(__, \mathsf{Cons}(__))$.[2]

4.2.2 Free variables and well-formedness

The definition of the free pattern variables $\mathit{FPV}(p)$ of a pattern is fairly straightforward; we build in a notion of well-formedness: if $\mathit{FPV}(p)$ is defined then p is *well-formed*: in every union $p_1 \sqcup p_2$ the free variables of p_1 and p_2 are the same, and no variable appears more than once (counting a variable appearing once in p_1 and once in p_2 as appearing once in $p_1 \sqcup p_2$).

Definition 4.1. $\mathit{FPV}(p)$ denotes the free pattern variables in p:

[2] Or-patterns are disallowed in the source program because pattern matching of unions is not deterministic: $p_1 \sqcup p_2 \xrightarrow{\sigma} v$ if $p_1 \xrightarrow{\sigma} v$ or $p_2 \xrightarrow{\sigma} v$. The operational semantics, defined in terms of $p \xrightarrow{\sigma} v$ for $p \Rightarrow e$, would become nondeterministic if p contained $\sqcup$s, and the combined type safety theorem (whose form we "inherit" from Chapter 2) is not useful if $e \mapsto e_1'$ and $e \mapsto e_2'$ does not imply $e_1' = e_2'$. It would be straightforward to resolve this by forcing matching of $p_1 \sqcup p_2$ to go left to right by adding a premise $v \notin p_1$ to the last rule of Figure 4.1, but that would require an inductive definition of $v \notin p_1$. Since or-patterns are not even part of Standard ML, we chose not to bother.

$$
\begin{array}{rcll}
FPV(x) & = & \{x\} & \\
FPV(()) & = & \{\} & \\
FPV((p_1, p_2)) & = & FPV(p_1) \cup FPV(p_2) & \text{if } FPV(p_1) \cap FPV(p_2) = \{\} \\
FPV(x \text{ as } p) & = & \{x\} \cup FPV(p) & \text{if } x \notin FPV(p) \\
FPV(c(p)) & = & FPV(p) & \\
FPV(_) & = & \{\} & \\
FPV(\{\}) & = & \{\} & \\
FPV(p_1 \sqcup p_2) & = & FPV(p_1) & \text{if } FPV(p_1) = FPV(p_2)
\end{array}
$$

In the remainder of this chapter, we **implicitly assume** that every pattern is well-formed.

4.2.3 Pattern matching

We inductively define a judgment $p \xrightarrow{\sigma} v$ that holds if σ applied to p yields v (Figure 4.1).[3] When σ is not of interest, it is convenient to write $v \in p$, read "v matches p":

Definition 4.2. $v \in p$ if and only if there exists σ such that $p \xrightarrow{\sigma} v$.

We extend the operational semantics from a single disjoint layer to sequential matching (Figure 4.2), introducing a "tiny step" relation $e \mapsto_M e'$ modeling sequential evaluation of match sequences ms. For example,

$$
\begin{array}{ll}
 & \textbf{case } \mathsf{Cons}(3, \mathsf{Nil}) \textbf{ of } \mathsf{Nil} \Rightarrow e_1 \mid \mathsf{Cons}(h, t) \Rightarrow e_2 \\
\mapsto_M & \textbf{case } \mathsf{Cons}(3, \mathsf{Nil}) \textbf{ of } \mathsf{Cons}(h, t) \Rightarrow e_2 \\
\mapsto_M & [3/h, \mathsf{Nil}/t]\; e_2
\end{array}
$$

This is necessary to avoid temporarily creating an ill-typed term: the intermediate term

$$\textbf{case } \mathsf{Cons}(3, \mathsf{Nil}) \textbf{ of } \mathsf{Cons}(h, t) \Rightarrow e_2$$

is nonexhaustive, and this would break type preservation. Rule ev-match$^+$ provides the transition from $\mapsto_M$ to $\mapsto_R$: if **case** v **of** $ms \mapsto_M^+ e$ then **case** v **of** $ms \mapsto_R e$, where $\mapsto_M^+$ denotes one or more applications of $\mapsto_M$.

There is no reduction rule that "falls off the end"—terms such as **case** $c_1()$ **of** $c_2() \Rightarrow e$ will get stuck:

$$\textbf{case } c_1() \textbf{ of } c_2() \Rightarrow e \;\;\mapsto\;\; \textbf{case } c_1() \textbf{ of } \cdot \;\; \not\mapsto$$

This is by design. In our type system, a program will pass typechecking only if all matches are exhaustive. In Standard ML, users are forced to either write nonexhaustive matches (and become used to the compiler crying wolf) or add spurious $_ \Rightarrow$ **raise** Match catch-alls (which clutter the code and are tempting to add without thinking, just to get rid of the warning). With refinements, the typechecker can determine that matches a simple typechecker would consider nonexhaustive, such as $x{:}\mathsf{red} \vdash$ **case** x **of** Red $_ \Rightarrow e$, are in fact exhaustive. Hence the warning can reasonably be made an error. In instances where exhaustiveness is due to an invariant not captured by type refinements, the user will have to add a catch-all, but such cases should be rare.

[3] This notation was chosen instead of $[\sigma]p = v$ because the latter strongly suggests that $[\sigma]p$ is a function, but σ and p do not uniquely determine v: consider $\sigma = \cdot$ and $p = c_1() \sqcup c_2()$.

$$\frac{}{() \xrightarrow{\cdot} ()} \qquad \frac{p_1 \xrightarrow{\sigma_1} v_1 \quad p_2 \xrightarrow{\sigma_2} v_2}{(p_1, p_2) \xrightarrow{\sigma_1,\sigma_2} (v_1, v_2)} \qquad \frac{p \xrightarrow{\sigma} v}{c(p) \xrightarrow{\sigma} c(v)}$$

$$\frac{}{_ \xrightarrow{\cdot} v} \qquad \frac{}{x \xrightarrow{v/x} v} \qquad \frac{p \xrightarrow{\sigma} v}{x \text{ as } p \xrightarrow{\sigma, v/x} v} \qquad \frac{p_1 \xrightarrow{\sigma} v}{p_1 \sqcup p_2 \xrightarrow{\sigma} v} \qquad \frac{p_2 \xrightarrow{\sigma} v}{p_1 \sqcup p_2 \xrightarrow{\sigma} v}$$

Figure 4.1: Definition of pattern matching

$$\frac{p \xrightarrow{\sigma} v}{\textbf{case } v \textbf{ of } p \Rightarrow e \mid ms \ \mapsto_M \ [\sigma]e} \text{ ev-match} \qquad \frac{v \notin p}{\textbf{case } v \textbf{ of } p \Rightarrow e \mid ms \ \mapsto_M \ \textbf{case } v \textbf{ of } ms} \text{ ev-no-match}$$

$$\frac{\textbf{case } v \textbf{ of } ms \mapsto_M^+ e}{\textbf{case } v \textbf{ of } ms \mapsto_R e} \text{ ev-match}^+$$

Figure 4.2: Operational semantics

4.2.4 Subtraction and intersection

A notion of pattern subtraction is needed to reason about sequential pattern matching. The result of pattern subtraction can be the empty pattern (as in $x - y$), or a union of patterns: if c_1, c_2, c_3 are the constructors of some datatype, $_ - c_1(_) = c_2(_) \sqcup c_3(_)$.

We do not formally define pattern subtraction; we require only that it satisfy the following property:

Property 4.3. *If* $p \xrightarrow{\sigma} v$ *and* $v \notin p'$ *then* $p - p' \xrightarrow{\sigma} v$.

If this property does not hold, type safety can be violated. Suppose e has type δ where δ has constructors c_1, c_2 and **case** e **of** $c_1(x) \Rightarrow e_1$. If $_ - c_1(x) = \{\}$, the case expression would be considered exhaustive, yet we could not make a transition when $e = c_2(v)$.

A consequence of the property is that $p - p'$ must 'prefer' the variables in p. For example, it cannot be the case that $(c_1(x) \sqcup c_2(x)) - c_1(y) = c_2(y)$, because then the two σs in the property would not be the same: $c_1(x) \sqcup c_2(x) \xrightarrow{v/x} c_2(v)$, but $c_2(y) \xrightarrow{v/y} c_2(v)$.

Likewise, we do not define pattern intersection $p_1 \cap p_2$, but it must satisfy Property 4.4.

Property 4.4. *If* $p_1 \xrightarrow{\sigma_1} v$ *and* $v \in p_2$ *then* $p_1 \cap p_2 \xrightarrow{\sigma_1} v$.

As with subtraction, type safety is threatened if this property fails to hold. A case arm $p \Rightarrow e$ is checked against the intersection of p and whatever pattern "remains" (initially the wildcard pattern). Suppose we have lists refined by empty and nonempty. If, absurdly, $_ \cap x = \text{Cons}(_)$, we can incorrectly derive (**case** e **of** $x \Rightarrow x$) $\downarrow$ nonempty, since only Cons(_) will be considered, leaving out the possibility that the value is Nil().

As with subtraction, the intersection property is written so that intersection must 'prefer' the pattern variables in the first pattern. Correspondingly, rule casearm has $p \cap p^*$ where p is the

pattern given by the user in a match $p \Rightarrow e$ and p^* is the pattern representing the space of remaining possible values. The pattern variables in p appear in the term e, so they should be preserved, while those in p^* are irrelevant.

Both properties are needed to prove Lemma 4.13.

4.3 Overview

At a high level, typechecking an expression **case** e **of** $p_1 \Rightarrow e_1 \mid ms$ is straightforward:

1. Synthesize a type A for e (the *scrutinee*).
2. Let $p := _$, the wildcard pattern.
3. Compute the intersection $p_1 \cap p$. In general, this produces a possibly empty union of patterns $p_{11} \sqcup \ldots \sqcup p_{1n}$.
4. For each disjunct p_{1k}, check e_1 under appropriate typing assumptions for pattern variables appearing in p_{1k}.
5. Let $p := p - p_1$.
6. Repeat steps 3–5 on the remaining matches in ms.
7. If p is nonempty, check if any values matching p are possible given A. If so, inform the user that the match is nonexhaustive.

In step 4, e_1 may need to be checked several times under varying assumptions. For example, assuming $c : (A_1 \to \delta(1)) \wedge (A_2 \to \delta(1))$ and $x : \delta(1)$, given **case** x **of** $c(y) \Rightarrow f(y)$ step 4 must check $f(y)$ first under the assumption that $y : A_1$ and then under the assumption that $y : A_2$. (In the earlier system, this was managed by rule $\wedge$-ct.)

Determining types for the pattern variables in a way that is both *sound* (if the scrutinee value matches the pattern, the matched values really do check against the types determined) and sufficiently *precise* (a notion we do not attempt to formalize[4]) is not trivial. A major confounding factor is the need to pass information up and down in x as p patterns, as in the example given earlier (p. 91). This precludes a simpler formulation via left rules [Dav05a, ch. 6]: in a system with Γ extended to permit pattern typings $p : A$, the derivation

$$\dfrac{\dfrac{\Gamma, x : \mathsf{badRoot} \vdash \ldots}{\Gamma, x : \mathsf{badRoot}, \mathsf{Empty} : \mathsf{badRoot} \vdash \ldots}}{\Gamma, (x \text{ as } \mathsf{Empty}) : \mathsf{badRoot} \vdash \ldots}$$

[4]Precision is a kind of completeness, but completeness with respect to what? We are not even prepared to claim we know completeness when we see it, though some things are clearly incomplete: it would be perfectly sound to use $\top$ for every pattern variable, but nothing useful would typecheck.

loses the improved type black for x.[5] Instead, we formulate a system based on *continuing judgments* of the form

$$\Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}$$

This is to be read "under context Γ, matching a value of type A against p yields information that, when 'passed' to $\mathcal{J}$, validates $\mathcal{J}$." A so-called *continuation* $\mathcal{J}$ is, in general, a sequence $\mathcal{J}_1 \;\rhd\; \dots \;\rhd\; \mathcal{J}_n$ where $\rhd$ is right associative; $\rhd$ may be loosely read as left-to-right function composition: $(\mathcal{J} \;\rhd\; \mathcal{J}')(-)$ acts as $\mathcal{J}'(\mathcal{J}(-))$. One of the simpler forms of a $\mathcal{J}$ is $\textsc{forgettype} \;\rhd\; e \downarrow C$, which, when 'passed' information—a type A and context Γ—drops the type and checks e (the body of some case arm) against C under Γ.

The rules analyze p to figure out what holds for all $v \in p$ of type A, producing both a new context Γ, Γ' (where Γ' types pattern variables encountered along the way) and an $A' \leq A$ (as in the earlier example, where $A = \mathsf{badRoot}$ and $A' = \mathsf{black}$). Both of these are 'passed' to $\mathcal{J}$ via a subderivation concluding

$$A' + \Gamma \vdash \mathcal{J}$$

which should be read "(under Γ) if the scrutinee is known to have type A', then $\mathcal{J}$ holds", the $+$ is punctuation. A typical derivation structure is

$$\frac{\begin{array}{c} A' + (\Gamma, \Gamma') \vdash \mathcal{J} \\ \vdots \end{array}}{\Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}}$$

where $A' \leq A$. The type A' can be thought of as being generated by the elided portion of the derivation, and then 'passed' to $\mathcal{J}$ along with the typing context, which has been extended with Γ'. More generally, derivations have the form

$$\frac{\begin{array}{ccc} A_1' + (\Gamma, \Gamma_1') \vdash \mathcal{J} & & A_n' + (\Gamma, \Gamma_n') \vdash \mathcal{J} \\ \vdots & \cdots & \vdots \end{array}}{\Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}}$$

where $A_k' \leq A_k$ for $k \in 1..n$.

For the simplest instance of this structure, consider a case arm $x \Rightarrow e$. First we give the rule $\mathsf{casearm}$.

$$\frac{\Gamma, \mathit{FPV}(p) \vdash e\ \mathsf{ok} \qquad \begin{array}{l} \Gamma \vdash (p \cap p^*) \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e \downarrow C) \\ \Gamma \vdash ((p^* - p) \downarrow A) \rhd ms \;\downarrow\; C \end{array}}{\Gamma \vdash (p^* \downarrow A) \rhd (p \Rightarrow e \mid ms) \;\downarrow\; C}\ \mathsf{casearm}$$

Say we have $p \cap p^* = x$ (for $x \Rightarrow e$). One of $\mathsf{casearm}$'s premises, $\Gamma \vdash x \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e \downarrow C)$, can be read "check pattern x against A, producing a context and a type; forget the type; check the

[5]The flavor of this fictional derivation is similar to the simplified pattern matching in Chapter 2, if one generalizes the $c(x) : \delta(i)$ assumption to $p : A$.

body e against C". In terms of the structure illustrated above, $\mathcal{J}$ is $\textsc{forgettype} \;\rhd\; e \downarrow C$ and $A' = A$.

$$\dfrac{\vdots \atop A + (\Gamma, x{:}A) \vdash \textsc{forgettype} \;\rhd\; e \downarrow C}{\Gamma \vdash x \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e \downarrow C)}\ \text{var-p}$$

Rule var-p (Figure 4.5) does not provide any new information about the scrutinee, so it immediately 'calls' its continuing judgment $\textsc{forgettype} \;\rhd\; e \downarrow C$ with the same type A and a context with $x{:}A$ added. The sole rule for $\textsc{forgettype} \;\rhd\; \mathcal{J}$ judgments simply forgets the type before the $+$ and 'calls' $\mathcal{J}$—which is here the garden-variety typing judgment $e \downarrow C$.

$$\dfrac{\dfrac{\Gamma, x{:}A \vdash e \downarrow C}{A + (\Gamma, x{:}A) \vdash \textsc{forgettype} \;\rhd\; e \downarrow C}\ \textsc{forgettype}\text{-}\mathcal{J}}{\Gamma \vdash x \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e \downarrow C)}\ \text{var-p}$$

The rules for x as p are key:

$$\dfrac{\Gamma \vdash p \downarrow A \;\rhd\; (\textsc{as}\ x \;\rhd\; \mathcal{J})}{\Gamma \vdash x \ \mathsf{as}\ p \downarrow A \;\rhd\; \mathcal{J}}\ \mathsf{as}\text{-p} \qquad \dfrac{A' + (\Gamma, x{:}A') \vdash \mathcal{J}}{A' + \Gamma \vdash \textsc{as}\ x \;\rhd\; \mathcal{J}}\ \textsc{as}\text{-p}$$

In these rules, A' may differ from A. In the derivation below, A (in the conclusion of as-p) is the initially known type badRoot, and A' (in rule AS-p) is black (where $\mathsf{black} \preceq \mathsf{badRoot}$):[6]

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma} \vdash \mathsf{Empty} : \mathsf{black} \qquad \Gamma \vdash \mathsf{black} \leq \mathsf{badRoot} \qquad \dfrac{\overbrace{\mathsf{black}}^{A'} + (\Gamma, x{:}\mathsf{black}) \vdash \mathcal{J}}{\mathsf{black} + \Gamma \vdash \textsc{as}\ x \;\rhd\; \mathcal{J}}\ \textsc{as}\text{-p}}{\mathsf{black} + \Gamma \vdash \textsc{con}\ \mathsf{Empty} \downarrow \mathsf{badRoot} \;\rhd\; (\textsc{as}\ x \;\rhd\; \mathcal{J})}\ \textsc{con}\text{-p}}{\vdots \atop \Gamma \vdash \mathsf{Empty} \downarrow \mathsf{badRoot} \;\rhd\; (\textsc{as}\ x \;\rhd\; \mathcal{J})}}{\Gamma \vdash x\ \mathsf{as}\ \mathsf{Empty} \downarrow \underbrace{\mathsf{badRoot}}_{A} \;\rhd\; \mathcal{J}}\ \mathsf{as}\text{-p}$$

With the judgment forms established, the rules deriving $\Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}$ are fairly easy to formulate. These rules, collected in Figure 4.5, are divided into two classes: pattern-directed rules that decompose p, and type-directed rules that decompose A. We have already seen rule var-p for variables and rule as-p for patterns of the form x as p. The rules for wildcards and unit patterns are similar to var-p but do not add to Γ:

$$\dfrac{A + \Gamma \vdash \mathcal{J}}{\Gamma \vdash _ \downarrow A \;\rhd\; \mathcal{J}}\ _\text{-p} \qquad \dfrac{\mathbf{1} + \Gamma \vdash \mathcal{J}}{\Gamma \vdash () \downarrow \mathbf{1} \;\rhd\; \mathcal{J}}\ ()\text{-p}$$

Since nothing can match the empty pattern $\{\}$, its rule has no premises. Likewise, given a pattern union $p_1 \sqcup p_2$, both p_1 and p_2 must be checked: $p_1 \sqcup p_2 \Rightarrow e$ is essentially the same as $p_1 \Rightarrow e$ and

[6] This derivation elides some detail via the nullary constructor Empty. Rule CON-p is discussed below.

$p_2 \Rightarrow e$.

$$\frac{}{\Gamma \vdash \{\} \downarrow A \;\rhd\; \mathcal{J}}\;\{\}\text{-p} \qquad \frac{\Gamma \vdash p_1 \downarrow A \;\rhd\; \mathcal{J} \qquad \Gamma \vdash p_2 \downarrow A \;\rhd\; \mathcal{J}}{\Gamma \vdash p_1 \sqcup p_2 \downarrow A \;\rhd\; \mathcal{J}}\;\sqcup\text{-p}$$

Some machinery is needed for pair patterns: given $\Gamma \vdash (p_1, p_2) \downarrow A_1 * A_2 \;\rhd\; \mathcal{J}$, we first analyze p_1, yielding subderivations $A'_{1k} + \Gamma, \Gamma'_{1k} \vdash \dots$. In each of these, we analyze p_2, resulting in sub-subderivations $A'_{2kj} + \Gamma, \Gamma'_{1k}, \Gamma'_{2kj} \vdash \dots$. Finally, the A'_{1k} and A'_{2kj} are combined to yield subderivations $(A'_{1k} * A'_{2kj}) + \Gamma, \Gamma'_{1k}, \Gamma'_{2kj} \vdash \mathcal{J}$. Doing this requires "administrative rules" similar to AS-p above. In a simple derivation involving these rules, a single subderivation results from analyzing p_1 and from analyzing p_2, as in the case where $p_1 = x_1$ and $p_2 = x_2$. In this case, the components are simply variables and provide no information, but in general $A'_{1k} \leq A_1$ and $A'_{2kj} \leq A_2$.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{(A_1 * A_2) + \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash \mathcal{J}}{A_2 + \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash \textsc{combine2}\ (A_1)(*) \;\rhd\; \mathcal{J}}\;\textsc{combine2}\text{-p}}{\Gamma, x_1{:}A_1 \vdash x_2 \downarrow A_2 \;\rhd\; (\textsc{combine2}\ (A_1)(*) \;\rhd\; \mathcal{J})}\;\text{var-p}}{A_1 + \Gamma, x_1{:}A_1 \vdash \textsc{combine}\ (*)(x_2 \downarrow A_2) \;\rhd\; \mathcal{J}}\;\textsc{combine}\text{-p}}{\Gamma \vdash x_1 \downarrow A_1 \;\rhd\; (\textsc{combine}\ (*)(x_2 \downarrow A_2) \;\rhd\; \mathcal{J})}\;\text{var-p}}{\Gamma \vdash (x_1, x_2) \downarrow A_1 * A_2 \;\rhd\; \mathcal{J}}\;(_,_)\text{-p}$$

Analysis of constructor patterns, in rule c-p,

$$\frac{\Gamma; c : \mathcal{S}(c); c(p) : \delta(i) \vdash \mathcal{J}}{\Gamma \vdash c(p) \downarrow \delta(i) \;\rhd\; \mathcal{J}}\;\text{c-p}$$

devolves to a judgment of the form $\Gamma; c : A^{con}; c(p) : C \vdash \mathcal{J}$, read "under context Γ, if c has type A^{con} and we have a scrutinee of type C matching a pattern $c(p)$ then $\mathcal{J}$ holds". The rules for this judgment in Figure 4.4 closely follow those in Chapter 2, with the exception that δS-ct devolves to a premise of the form $\Gamma, i \doteq i' \vdash p \downarrow A \;\rhd\; ((\textsc{con}\ c \downarrow \delta(i)) \;\rhd\; \mathcal{J})$, which analyzes p, arriving at a subderivation (in general, zero or more subderivations) of the form $\dots \vdash \textsc{con}\ c \downarrow \delta(i) \;\rhd\; \mathcal{J}$. The relevant administrative rule CON-p refers *again* to the type of c (which was already broken down by the rules in Figure 4.4) in order to get a more accurate type of $c(p)$: the analysis of p has resulted in a possibly improved type B, so if $c : C_1 \to C_2$ with $B \leq C_1$ and $C_2 \leq A$, we have $c(p) : C_2$. In the derivation structure we have established, this deduction takes the form of a premise $C_2 + \Gamma \vdash \mathcal{J}$.

$$\frac{\overline{\Gamma} \vdash c : C_1 \to C_2 \qquad \Gamma \vdash B \leq C_1 \qquad \Gamma \vdash C_2 \leq A \qquad C_2 + \Gamma \vdash \mathcal{J}}{B + \Gamma \vdash \textsc{con}\ c \downarrow A \;\rhd\; \mathcal{J}}\;\textsc{con}\text{-p}$$

The type-directed rules—for $\wedge$, $\bot$, $\vee$, Σ and $\otimes$—are fairly straightforward. The rule for $\wedge$ uses COMBINE continuing judgments, similar to the rule for $*$. There are no rules for $\top$, Π, and $\supset$; to understand why, see Section 4.7.

Exhaustiveness checking involves several rules. If the current pattern space is p^*, a sequence of empty matches, representing the end of a case expression, checks provided that no value of type A can actually match p^*. The premise of emptyms is read "under Γ, if the scrutinee (of type A) matches p^*, something impossible happens"; the premise is derivable only if it *cannot* be the case that the scrutinee matches p^*.

$$\dfrac{\Gamma \vdash p^* \downarrow A \rhd \textsc{impossible}}{\Gamma \vdash (p^* \downarrow A) \rhd \cdot \downarrow C}\ \mathsf{emptyms}$$

If $p^* = \{\}$, no value can possibly match p^*. This is captured by rule {}-p, which derives $\Gamma \vdash \{\} \downarrow A \ \rhd\ \mathcal{J}$ for any $\mathcal{J}$, including IMPOSSIBLE:

$$\dfrac{}{\Gamma \vdash \{\} \downarrow A \ \rhd\ \mathcal{J}}\ \{\}\text{-p}$$

This rule permits the following derivation, which can be used to show that *syntactically* exhaustive matches are exhaustive.

$$\dfrac{\dfrac{}{\Gamma \vdash \{\} \downarrow A \ \rhd\ \textsc{impossible}}\ \{\}\text{-p}}{\Gamma \vdash (\{\} \downarrow A) \ \rhd\ \cdot \downarrow C}\ \mathsf{emptyms}$$

With emptyms and {}-p we have the same conservative exhaustiveness checking found in Standard ML. However, we have additional rules bringing datasort and index refinements into play, so that even if p^* is syntactically nonempty ($p^* \neq \{\}$) we can show that the matches are exhaustive, as in this example with a nonempty datasort refinement of lists (where $\mathcal{S}(\mathsf{Nil}) = \mathbf{1} \to \mathsf{empty}$): given a value of type nonempty, it is impossible that the value is $\mathsf{Nil}()$, so the matches are exhaustive even though the space of remaining patterns p^* is not empty.

$$\dfrac{\dfrac{\dfrac{\mathsf{nonempty} \not\preceq \mathsf{empty}}{\Gamma; \mathsf{Nil} : \mathbf{1} \to \mathsf{empty}; \mathsf{Nil}() : \mathsf{nonempty} \vdash \textsc{impossible}}\ \delta\text{F-ct}}{\Gamma \vdash \mathsf{Nil}() \downarrow \mathsf{nonempty} \ \rhd\ \textsc{impossible}}\ \text{c-p}}{\Gamma \vdash (\mathsf{Nil}() \downarrow \mathsf{nonempty}) \ \rhd\ \cdot \downarrow C}\ \mathsf{emptyms}$$

The final rule, contra-$\mathcal{J}$, that can (ultimately) derive the premise of emptyms is neither pattern- or type-directed. contra-$\mathcal{J}$, analogous to rule contra, allows one to conclude any $\ldots \vdash \mathcal{J}$ if $\overline{\Gamma} \models \bot$.

$$\dfrac{\Gamma \models \bot}{B + \Gamma \vdash \mathcal{J}}\ \text{contra-}\mathcal{J}$$

For example, with an index refinement $\mathcal{S}(\mathsf{Nil}) = \mathbf{1} \to \mathsf{list}(0)$, rule contra-$\mathcal{J}$ can "short-circuit" the derivation:

$$\dfrac{\dfrac{\mathsf{list} \preceq \mathsf{list} \qquad \dfrac{\dfrac{0 \doteq 2 \models \bot}{\mathbf{1} + (\Gamma, 0 \doteq 2) \vdash \textsc{con}\ c \downarrow \mathsf{list}(2) \ \rhd\ \textsc{impossible}}\ \text{contra-}\mathcal{J}}{\Gamma, 0 \doteq 2 \vdash () \downarrow \mathbf{1} \ \rhd\ (\textsc{con}\ c \downarrow \mathsf{list}(2) \ \rhd\ \textsc{impossible})}\ ()\text{-p}}{\dfrac{\Gamma; \mathsf{Nil} : \mathbf{1} \to \mathsf{list}(0); \mathsf{Nil}() : \mathsf{list}(2) \vdash \mathcal{J}}{\Gamma \vdash \mathsf{Nil}() \downarrow \mathsf{list}(2) \ \rhd\ \textsc{impossible}}\ \text{c-p}}\ \delta\text{S-ct}}{\Gamma \vdash \mathsf{Nil}() \downarrow \mathsf{list}(2) \ \rhd\ \cdot \downarrow C}\ \mathsf{emptyms}$$

$$
\begin{array}{rrcl}
\text{Patterns} & p & ::= & x \mid () \mid (p_1, p_2) \mid x \text{ as } p \mid c(p) \mid _ \mid \{\} \mid p_1 \sqcup p_2 \\
\\
\textsc{combine}\text{ operators} & \Theta & ::= & \wedge \mid * \\
\text{Continuing judgments} & \mathcal{J} & ::= & \textsc{forgettype} \ \rhd \ e \downarrow A \\
& & \mid & \textsc{forgettype} \ \rhd \ e : A \\
& & \mid & \textsc{impossible} \\
& & \mid & \textsc{as}\ x \ \rhd \ \mathcal{J} \\
& & \mid & \textsc{con}\ c \downarrow A \ \rhd \ \mathcal{J} \\
& & \mid & \textsc{combine}\ (\Theta)(p \downarrow A) \ \rhd \ \mathcal{J} \\
& & \mid & \textsc{combine2}\ (A)(\Theta) \ \rhd \ \mathcal{J}
\end{array}
$$

$$
\begin{array}{rcl}
[\sigma](\textsc{forgettype} \ \rhd \ e : A) & = & \textsc{forgettype} \ \rhd \ [\sigma]e : [\sigma]A \\
[\sigma]\ (\textsc{as}\ x \ \rhd \ \mathcal{J}) & = & \textsc{as}\ x \ \rhd \ [\sigma]\mathcal{J} \\
[\sigma]\ (\textsc{con}\ c \downarrow A \ \rhd \ \mathcal{J}) & = & \textsc{con}\ c \downarrow [\sigma]A \ \rhd \ [\sigma]\mathcal{J} \\
[\sigma]\ (\textsc{combine}\ (\Theta)(p \downarrow A) \ \rhd \ \mathcal{J}) & = & \textsc{combine}\ (\Theta)(p \downarrow [\sigma]A) \ \rhd \ [\sigma]\mathcal{J} \\
[\sigma]\ (\textsc{combine2}\ (A)(\Theta) \ \rhd \ \mathcal{J}) & = & \textsc{combine2}\ ([\sigma]A)(\Theta) \ \rhd \ [\sigma]\mathcal{J} \\
[\sigma]\ \textsc{impossible} & = & \textsc{impossible}
\end{array}
$$

Figure 4.3: Grammar

4.3.1 Case, match, and constructor typing

The rules for typing **case** expressions, matches ms, and terms with a constructor on the left (in previous chapters, the judgment form used was $\Gamma; c : A^{con}; c(x) : B \vdash e \downarrow C$) are fairly similar to those in previous chapters. See Figure 4.4.

The form of an ms-checking judgment is generalized from $\Gamma \vdash ms \downarrow_B C$ to

$$\Gamma \vdash (p^* \downarrow B) \ \rhd \ ms \downarrow C$$

where p^* is the remaining pattern space (initially the wildcard _, in rule δE), which is subtracted from as the derivation moves along the sequence of matches. The new casearm checks the first case arm $p \Rightarrow e$ with the intersection of p and p^*, then checks the remaining case arms with p subtracted out.[7]

The earlier judgment form $\Gamma; c : A^{con}; c(x) : B \vdash e \downarrow C$ is generalized to $\Gamma; c : A^{con}; c(p) : B \vdash \mathcal{J}$, replacing the variable x with an arbitrary nested pattern p, and replacing $e \downarrow C$ with $\mathcal{J}$ to allow the resulting obligations to be 'passed' to either FORGETTYPE $\rhd$ $e \downarrow C$ or a pattern checking judgment (if $c(p)$ is itself nested). Note that δS-ct's premise

$$\Gamma, i \doteq i' \vdash p \downarrow A \ \rhd \ (\textsc{con}\ c \downarrow \delta(i) \ \rhd \ \mathcal{J})$$

[7]The premise $\Gamma, FPV(p) \vdash e$ ok ensures that all variables free in e are known, either in Γ or as pattern variables in p. This premise is not actually used anywhere; it is here only to bolster the argument that the work in Chapter 5, which is based on the simple pattern matching of earlier chapters, could be readily extended to handle the full pattern language presented here. The simple pattern matching system includes a similar premise in its version of δF-ct; putting a premise in our δF-ct would be problematic because there is no immediate way to "dig out" the e at the end of $\mathcal{J}$.

$\boxed{\Gamma; c : A^{con}; c(p) : B \vdash \mathcal{J}}$

$$\frac{\delta \preceq \delta' \qquad \Gamma, i \doteq i' \vdash p \downarrow A \;\rhd\; (\textsc{con}\ c \downarrow \delta(i) \;\rhd\; \mathcal{J})}{\Gamma; c : A \to \delta(i); c(p) : \delta'(i') \vdash \mathcal{J}}\ \delta\text{S-ct} \qquad \frac{\delta \not\preceq \delta'}{\Gamma; c : A \to \delta(i); c(p) : \delta'(i') \vdash \mathcal{J}}\ \delta\text{F-ct}$$

$$\frac{\Gamma; c : A_1; c(p) : B \vdash \mathcal{J} \quad \Gamma; c : A_2; c(p) : B \vdash \mathcal{J}}{\Gamma; c : A_1 \wedge A_2; c(p) : B \vdash \mathcal{J}}\ \wedge\text{-ct} \qquad \frac{\Gamma, a{:}\gamma; c : A; c(p) : B \vdash \mathcal{J}}{\Gamma; c : \Pi a{:}\gamma.\, A; c(p) : B \vdash \mathcal{J}}\ \Pi\text{-ct}$$

$$\frac{\Gamma, P; c : A; c(p) : B \vdash \mathcal{J}}{\Gamma; c : (P \supset A); c(p) : B \vdash \mathcal{J}}\ \supset\text{-ct}$$

$\boxed{\Gamma \vdash (p^* \downarrow A) \;\rhd\; ms \;\downarrow\; C}$

$$\frac{\Gamma, \mathit{FPV}(p) \vdash e\ \mathsf{ok} \qquad \begin{array}{l}\Gamma \vdash (p \cap p^*) \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e \downarrow C) \\ \Gamma \vdash ((p^* - p) \downarrow A) \rhd ms \;\downarrow\; C\end{array}}{\Gamma \vdash (p^* \downarrow A) \rhd (p \Rightarrow e \mid ms) \;\downarrow\; C}\ \mathsf{casearm}$$

$$\frac{\Gamma \vdash p^* \downarrow A \rhd \textsc{impossible}}{\Gamma \vdash (p^* \downarrow A) \rhd \cdot \downarrow C}\ \mathsf{emptyms}$$

$\boxed{\Gamma \vdash e \downarrow C}$ *Replaces the earlier* δE *(Chapter 3)*

$$\frac{\Gamma \vdash e \uparrow A \qquad \Gamma \vdash (_ \downarrow A) \rhd ms \;\downarrow\; C}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ ms \downarrow C}\ \delta\text{E}$$

Figure 4.4: Typing rules for left-constructor judgments, matches, and **case**

analyzes p to obtain a (hopefully smaller) type A', which rule CON-p will use to obtain some $\delta' \preceq \delta$. These rules are otherwise substantially similar to the earlier rules—compare Figures 4.4 and 3.3 (p. 58).

4.4 Type assignment version of the system

The system presented thus far checks case arms against a type $(e \downarrow C)$, conforming to the simple tridirectional system of Chapter 3. In order to prove soundness and a substitution lemma, we formulate type assignment versions of the rules in Figure 4.4. The left-constructor typing rules speak only of an unspecified $\mathcal{J}$ and so require no changes at all. The type assignment versions of rules casearm, emptyms and FORGETTYPE-$\mathcal{J}$ simply have ": C" instead of "$\downarrow$ C" in their premises and conclusions. The pattern typing rules in Figure 4.5, like the left-constructor rules, speak only of some $\mathcal{J}$ and likewise require no changes. In particular, they keep their $p \downarrow A$ judgment form, as

$\boxed{\Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}}$ $\boxed{B + \Gamma \vdash \mathcal{J}}$

Concluding rules:

$$\frac{\Gamma \vdash e \downarrow C}{A + \Gamma \vdash \textsc{forgettype} \;\rhd\; e \downarrow C}\ \textsc{forgettype}\text{-}\mathcal{J} \qquad \frac{\Gamma \models \bot}{B + \Gamma \vdash \mathcal{J}}\ \mathsf{contra}\text{-}\mathcal{J}$$

Type-directed rules:

$$\frac{\Gamma \vdash p \downarrow A_1 \;\rhd\; (\textsc{combine}\ (\wedge)(p \downarrow A_2) \;\rhd\; \mathcal{J})}{\Gamma \vdash p \downarrow A_1 \wedge A_2 \;\rhd\; \mathcal{J}}\ \wedge\text{-p}$$

$$\frac{}{\Gamma \vdash p \downarrow \bot \;\rhd\; \mathcal{J}}\ \bot\text{-p} \qquad \frac{\Gamma \vdash p \downarrow A_1 \;\rhd\; \mathcal{J} \qquad \Gamma \vdash p \downarrow A_2 \;\rhd\; \mathcal{J}}{\Gamma \vdash p \downarrow A_1 \vee A_2 \;\rhd\; \mathcal{J}}\ \vee\text{-p}$$

$$\frac{\Gamma, a{:}\gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}}{\Gamma \vdash p \downarrow \Sigma a{:}\gamma.\, A \;\rhd\; \mathcal{J}}\ \Sigma\text{-p} \qquad \frac{\Gamma, P \vdash p \downarrow A \;\rhd\; \mathcal{J}}{\Gamma \vdash p \downarrow P \otimes A \;\rhd\; \mathcal{J}}\ \otimes\text{-p}$$

Pattern-directed rules:

$$\frac{A + (\Gamma, x{:}A) \vdash \mathcal{J}}{\Gamma \vdash x \downarrow A \;\rhd\; \mathcal{J}}\ \mathsf{var}\text{-p} \qquad \frac{A + \Gamma \vdash \mathcal{J}}{\Gamma \vdash _ \downarrow A \;\rhd\; \mathcal{J}}\ _\text{-p} \qquad \frac{\mathbf{1} + \Gamma \vdash \mathcal{J}}{\Gamma \vdash () \downarrow \mathbf{1} \;\rhd\; \mathcal{J}}\ ()\text{-p}$$

$$\frac{(A_1\ \Theta\ A_2) + \Gamma \vdash \mathcal{J}}{A_2 + \Gamma \vdash \textsc{combine2}\ (A_1)(\Theta) \;\rhd\; \mathcal{J}}\ \textsc{combine2}\text{-p}$$

$$\frac{\Gamma \vdash p_2 \downarrow A_2 \;\rhd\; (\textsc{combine2}\ (A_1)(\Theta) \;\rhd\; \mathcal{J})}{A_1 + \Gamma \vdash \textsc{combine}\ (\Theta)(p_2 \downarrow A_2) \;\rhd\; \mathcal{J}}\ \textsc{combine}\text{-p}$$

$$\frac{\Gamma \vdash p_1 \downarrow A_1 \;\rhd\; (\textsc{combine}\ (*)(p_2 \downarrow A_2) \;\rhd\; \mathcal{J})}{\Gamma \vdash (p_1, p_2) \downarrow A_1 * A_2 \;\rhd\; \mathcal{J}}\ (_,_)\text{-p}$$

$$\frac{\Gamma \vdash p \downarrow A \;\rhd\; (\textsc{as}\ x \;\rhd\; \mathcal{J})}{\Gamma \vdash x\ \mathsf{as}\ p \downarrow A \;\rhd\; \mathcal{J}}\ \mathsf{as}\text{-p} \qquad \frac{A + (\Gamma, x{:}A) \vdash \mathcal{J}}{A + \Gamma \vdash \textsc{as}\ x \;\rhd\; \mathcal{J}}\ \textsc{as}\text{-p}$$

$$\frac{\overline{\Gamma} \vdash c : C_1 \to C_2 \qquad \Gamma \vdash B \leq C_1 \qquad \Gamma \vdash C_2 \leq A \qquad C_2 + \Gamma \vdash \mathcal{J}}{B + \Gamma \vdash \textsc{con}\ c \downarrow A \;\rhd\; \mathcal{J}}\ \textsc{con}\text{-p}$$

$$\frac{\Gamma; c : \mathcal{S}(c); c(p) : \delta(i) \vdash \mathcal{J}}{\Gamma \vdash c(p) \downarrow \delta(i) \;\rhd\; \mathcal{J}}\ \text{c-p}$$

$$\frac{}{\Gamma \vdash \{\} \downarrow A \;\rhd\; \mathcal{J}}\ \{\}\text{-p} \qquad \frac{\Gamma \vdash p_1 \downarrow A \;\rhd\; \mathcal{J} \qquad \Gamma \vdash p_2 \downarrow A \;\rhd\; \mathcal{J}}{\Gamma \vdash p_1 \sqcup p_2 \downarrow A \;\rhd\; \mathcal{J}}\ \sqcup\text{-p}$$

Figure 4.5: Pattern typing rules

the logic of pattern typing is the same.

We briefly state soundness and completeness theorems corresponding to Theorem 3.3 and Corollary 3.16, eliding a full statement for the various new judgment forms and the proofs, which closely follow those of the earlier theorems: type annotations cannot appear in patterns and the pattern checking rules do not examine terms.

Theorem 4.5 (Soundness (Type Assignment))**.** *If* $\Gamma \vdash e \downarrow C$ *or* $\Gamma \vdash e \uparrow C$ *(in the system presented thus far) then* $\Gamma \vdash |e| : C$ *(in the type assignment version of that system).*

$|e|$ is the erasure of type annotations from e.

Theorem 4.6 (Completeness (Type Assignment))**.** *If* $\Gamma \vdash e : C$ *(in the type assignment version of the system presented) then* $\Gamma \vdash e' \downarrow C$ *(in the system presented) where* $e' \sqsupseteq e$.

All judgments and derivations in the remainder of this chapter are in the type assignment system.

4.4.1 Substitution

Definition 4.7 (Derivation measure)**.** Let $\mathcal{D}' \prec \mathcal{D}$ (resp. $\mathcal{D}' \preceq \mathcal{D}$) if and only if the height of $\mathcal{D}'$ is less than (resp. less than or equal to) the height of $\mathcal{D}$, considering every subtree deriving $\ldots \vdash \text{FORGETTYPE} \ldots$ to have height 1.

By counting subderivations concluding term typing judgments (as opposed to pattern typing judgments) as all being the same size, the measure allows us to apply the induction hypothesis to the result of substituting values for pattern variables in a subderivation. Simply counting the number of pattern checking rules applied in the whole derivation does not work, since the subderivation typing a substituted value v may have a **case** inside a λ inside v.

Lemma 4.8 (Substitution)**.** *If* $\Gamma' \vdash \sigma : \Gamma$ *then:*

(1) If $\mathcal{D} :: \Gamma \vdash p \downarrow A \;\rhd\; \mathcal{J}$ *then* $\mathcal{D}' :: \Gamma' \vdash p \downarrow [\sigma]A \;\rhd\; [\sigma]\mathcal{J}$.

(2) If $\mathcal{D} :: B + \Gamma \vdash \mathcal{J}$ *then* $\mathcal{D}' :: [\sigma]B + \Gamma' \vdash [\sigma]\mathcal{J}$.

(3) (As Lemma 2.14.)

(4) If $\mathcal{D} :: \Gamma; c : A^{con}; c(p) : B \vdash \mathcal{J}$ *then* $\mathcal{D}' :: \Gamma'; c : [\sigma]A^{con}; c(p) : [\sigma]B \vdash [\sigma]\mathcal{J}$.

where $\mathcal{D}' \preceq \mathcal{D}$.

Proof. Part (1):

- **Case** var-p: $\mathcal{D} :: \dfrac{A + (\Gamma, x{:}A) \vdash \mathcal{J}'}{\Gamma \vdash x \downarrow A \;\rhd\; \mathcal{J}'}$

$\Gamma' \vdash \sigma : \Gamma$	Given
$\Gamma', x{:}[\sigma]A \vdash \sigma : \Gamma$	By Lemma 2.13
$\Gamma', x{:}[\sigma]A \vdash x : [\sigma]A$	By var
$\Gamma', x{:}[\sigma]A \vdash (\sigma, x/x) : (\Gamma, x{:}A)$	By pvar-σ
$[\sigma]A + (\Gamma', x{:}[\sigma]A) \vdash [\sigma]\mathcal{J}'$	By IH(2)
$\Gamma' \vdash x \downarrow [\sigma]A \;\rhd\; [\sigma]\mathcal{J}'$	By var-p

- **Case** AS-p: $\mathcal{D} :: \dfrac{A + (\Gamma, x{:}A) \vdash \mathcal{J}'}{A + \Gamma \vdash \text{AS } x \;\rhd\; \mathcal{J}'}$

$$
\begin{array}{rll}
\Gamma' \vdash \sigma : \Gamma & & \text{Given} \\
\Gamma', x{:}[\sigma]A \vdash \sigma : \Gamma & & \text{By Lemma 2.13} \\
\Gamma', x{:}[\sigma]A \vdash x : [\sigma]A & & \text{By var} \\
\Gamma', x{:}[\sigma]A \vdash (\sigma, x/x) : (\Gamma, x{:}A) & & \text{By pvar-}\sigma \\
[\sigma]A + (\Gamma', x{:}[\sigma]A) \vdash [\sigma]\mathcal{J}' & & \text{By IH(2)} \\
\Gamma' \vdash \text{AS } x \;\rhd\; [\sigma]\mathcal{J}' & & \text{By AS-p}
\end{array}
$$

Part (2): In the FORGETTYPE-$\mathcal{J}$ case, use part (3).

The proof of part (3) follows that of Lemma 2.14, except that in the δE case, part (1) must be used.

The proof of part (4) is straightforward, using part (1) in the δS-ct case. □

4.5 Lemmas for soundness

To prove soundness, we need two lemmas related to constructors:

- Lemma 4.9 says that given $\Gamma; c : A^{con}; c(p) : \delta(i) \vdash \mathcal{J}$ where $A_1 \to \delta'(i')$ is a 'component' of A^{con}, there exists a smaller derivation of $\Gamma \vdash p \downarrow A_1 \;\rhd\; (\text{CON } c \downarrow \delta'(i') \;\rhd\; \mathcal{J})$. It plays part of the role that Lemma 2.19 did in Chapter 2.

- Lemma 4.10 is an easy inversion lemma on derivations of $\ldots \vdash \text{CON } c \downarrow A$; note that the derivation yielded by Lemma 4.9 contains such derivations.

Lemma 4.9. *If $\mathcal{D} :: \Gamma; c : A^{con}; c(p) : \delta(i) \vdash \mathcal{J}$ where $\Gamma \vdash A^{con} \uparrow A_1 \to \delta'(i')$ and $\Gamma \vdash \delta'(i') \le \delta(i)$, then there exists $\mathcal{D}' \prec \mathcal{D}$ where $\mathcal{D}' :: \Gamma \vdash p \downarrow A_1 \;\rhd\; (\text{CON } c \downarrow \delta'(i') \;\rhd\; \mathcal{J})$.*

Proof. By induction on $\mathcal{D}$ with the measure $\prec$ (Definition 4.7).

Note that $\mathcal{D}$ includes a smaller derivation $\ldots \vdash \mathcal{J}$ for each "component" of A^{con}, while it follows from $\Gamma \vdash A^{con} \uparrow A_1 \to \delta'(i')$ that $A_1 \to \delta'(i')$ is one of those components.

- **Case** δS-ct: $\mathcal{D} :: \dfrac{\delta'' \preceq \delta' \qquad \Gamma, i'' \doteq i' \vdash p \downarrow A_1'' \;\rhd\; (\text{CON } c \downarrow \delta''(i'')) \;\rhd\; \mathcal{J})}{\Gamma; c : A_1'' \to \delta''(i''); c(p) : \delta'(i') \vdash \mathcal{J}}$

  By inversion on $\Gamma \vdash A_1'' \to \delta''(i'') \uparrow A_1 \to \delta'(i')$, we have $A_1'' \to \delta''(i'') = A_1 \to \delta'(i')$, from which $A_1'' = A_1$, $\delta'' = \delta'$, $i'' = i'$ follow.

  Thus the second premise (subderivation), which is the desired $\mathcal{D}'$, is

$$\Gamma, i' \doteq i' \vdash p \downarrow A_1 \;\rhd\; (\text{CON } c \downarrow \delta'(i')) \;\rhd\; \mathcal{J})$$

  We have $\Gamma \models [\Gamma/\Gamma]\,(i' \doteq i')$ for all i', so by prop-σ, $\Gamma \vdash \Gamma/\Gamma : \Gamma, i' \doteq i'$, which with Lemma 4.8 yields $\Gamma \vdash p \downarrow A_1 \;\rhd\; (\text{CON } c \downarrow \delta'(i')) \;\rhd\; \mathcal{J})$.

- **Case** δF-ct: The premise is $\delta'' \not\preceq \delta'$. Yet inversion (as in the δS-ct case) yields $\delta'' = \delta'$. Thus we have $\delta' \not\preceq \delta'$, yet $\preceq$ is assumed to be reflexive, so we have a contradiction. Rule δF-ct could not have been used.

- **Case** $\wedge$-ct: We have $A = B_1 \wedge B_2$. By inversion on $\Gamma \vdash (B_1 \wedge B_2) \uparrow A_1 \to \delta'(i')$, either $\Gamma \vdash B_1 \uparrow A_1 \to \delta'(i')$ or $\Gamma \vdash B_2 \uparrow A_1 \to \delta'(i')$. Suppose the former (the second case is symmetric). The result follows by the IH.

- **Case** Π-ct: $\mathcal{D} :: \dfrac{\Gamma, a{:}\gamma; c : A_0; c(p) : \delta'(i') \vdash \mathcal{J}}{\Gamma; c : \Pi a{:}\gamma.\, A_0; c(p) : \delta'(i') \vdash \mathcal{J}}$

  By inversion on $\Gamma \vdash \Pi a{:}\gamma.\, A_0 \uparrow A_1 \to \delta'(i')$, there exists j such that $\Gamma \vdash [j/a]A_0 \uparrow A_1 \to \delta'(i')$ and $\overline{\Gamma} \vdash j : \gamma$.

  By Lemma 4.8, $\Gamma; c : [j/a]A_0; c(p) : \delta(i) \vdash \mathcal{J}$. (Note that a cannot be free in i or $\mathcal{J}$.) The result follows by IH.

- **Case** $\supset$-ct: Similar to the Π-ct case, using prop-σ in the manner of the δS-ct case.

□

Lemma 4.10. *If $\mathcal{D} :: B + \Gamma \vdash \textsc{con}\ c \downarrow A \ \rhd\ \mathcal{J}$ then $\mathcal{D}' :: C_2 + \Gamma \vdash \mathcal{J}$ where $\mathcal{D}' \prec \mathcal{D}$ and*

$$\overline{\Gamma} \vdash c : C_1 \to C_2, \qquad \Gamma \vdash B \leq C_1, \qquad \Gamma \vdash C_2 \leq A.$$

Proof.

	$B + \Gamma \vdash \textsc{con}\ c \downarrow A \ \rhd\ \mathcal{J}$	Given
☞	$\exists C_2.\ \overline{\Gamma} \vdash c : C_1 \to C_2$ and $\Gamma \vdash B \leq C_1$ and $\Gamma \vdash C_2 \leq A$	By inversion
☞	$\mathcal{D}' ::\ C_2 + \Gamma \vdash \mathcal{J}$	By inversion

□

4.6 Soundness

The soundness theorem (4.11) says that given $\vdash p \downarrow A \ \rhd\ \mathcal{J}$ and a value $\vdash v : A$, where $p \xrightarrow{\sigma} v$, there exists a smaller derivation of $A' + \cdot \vdash [\sigma]\mathcal{J}$ where $A' \leq A$ and $\vdash v : A'$. Recall the typical derivation structure discussed earlier:

$$\mathcal{D} :: \dfrac{\begin{array}{ccc} A'_1 + \Gamma'_1 \vdash \mathcal{J} & & A'_n + \Gamma'_n \vdash \mathcal{J} \\ \vdots & \cdots & \vdots \end{array}}{\cdot \vdash p \downarrow A \ \rhd\ \mathcal{J}}$$

Roughly, the soundness theorem takes one from the root derivation $\mathcal{D}$ to an appropriate $A'_i + \Gamma'_i \vdash \mathcal{J}$ derivation, that is, one such that $\vdash v : A'_i$. This is not quite accurate: the theorem's result is a derivation of $A' + \cdot \vdash [\sigma]\mathcal{J}$, in which the typing context is empty. To obtain a derivation with the empty context, the proof of the theorem uses inversion on $p \xrightarrow{\sigma} v$ with Lemma 4.8 to substitute

for pattern variables in Γ_i', and value definiteness (since $\vdash v : A$ is given) to substitute for index variables arising from rules such as Σ-p.

Note that if $\mathcal{J}$ is simply $\textsc{forgettype} \;\rhd\; e : C$—as it is in the premise of rule casearm—the theorem yields $\vdash \textsc{forgettype} \;\rhd\; [\sigma]e : C$. Only $\textsc{forgettype}$-$\mathcal{J}$ can derive such a judgment; inversion yields $\vdash [\sigma]e : C$, corresponding to reduction of a case arm $p \Rightarrow e$ to $[\sigma]e$ given $p \xrightarrow{\sigma} v$.

Theorem 4.11 (Soundness of pattern checking)**.** *For all $\mathcal{J}$, if $\mathcal{D} :: \cdot \vdash p \downarrow A \;\rhd\; \mathcal{J}$*
and $\cdot \vdash v : A$
and $p \xrightarrow{\sigma} v$
then
there exists $\mathcal{D}' :: A' + \cdot \vdash [\sigma]\mathcal{J}$
where $\mathcal{D}' \prec \mathcal{D}$ (Definition 4.7) and
(i) $\cdot \vdash A' \leq A$,
(ii) $\cdot \vdash v : A'$.

Proof. By induction on the derivation $\mathcal{D}$ with measure $\prec$ (Definition 4.7).

- **Case** var-p: $\mathcal{D} :: \dfrac{A + (\cdot, x{:}A) \vdash \mathcal{J}}{\cdot \vdash x \downarrow A \;\rhd\; \mathcal{J}}$

  $p = x$. Let $A' = A$.

  By empty-σ and pvar-σ, $\cdot \vdash v/x : x{:}A$. By Lemma 4.8, $A + \cdot \vdash [v/x]\mathcal{J}$. By inversion on $x \xrightarrow{\sigma} v$ we have $\sigma = v/x$, so in fact $A + \cdot \vdash [\sigma]\mathcal{J}$, which was to be shown.

  Parts (i)–(ii) are easily satisfied: (i) $\cdot \vdash A' \leq A$ by Lemma 2.9; (ii) $\cdot \vdash v : A'$ is the same as $\cdot \vdash v : A$ which was given.

- **Case** __-p: Similar to the var-p case, but simpler.

- **Case** ()-p: $p = ()$ and $A = \mathbf{1}$. Similar to the var-p case, but simpler.

- **Case** (_, _)-p: $p = (p_1, p_2)$ and $A = A_1 * A_2$.

$$
\begin{array}{ll}
\cdot \vdash p_1 \downarrow A_1 \;\rhd\; (\textsc{combine}\ (*)(p_2 \downarrow A_2) \;\rhd\; \mathcal{J}) & \text{Subderivation} \\
\cdot \vdash v : A_1 * A_2 & \text{Given} \\
v = (v_1, v_2) \quad \text{and} \quad \cdot \vdash v_1 : A_1 \quad \text{and} \quad \cdot \vdash v_2 : A_2 & \text{By Lemma 2.18} \\
(p_1, p_2) \xrightarrow{\sigma} v & \text{Given} \\
\sigma = \sigma_1, \sigma_2 \quad \text{and} \quad p_1 \xrightarrow{\sigma_1} v_1 \quad \text{and} \quad p_2 \xrightarrow{\sigma_2} v_2 & \text{By inversion} \\
A_1' + \cdot \vdash \textsc{combine}\ (*)(p_2 \downarrow A_2) \;\rhd\; [\sigma_1]\mathcal{J} & \\
\cdot \vdash A_1' \leq A_1 \quad \text{and} \quad \vdash v_1 : A_1' & \text{By IH} \\
\cdot \vdash p_2 \downarrow A_2 \;\rhd\; (\textsc{combine2}\ (A_1')(*) \;\rhd\; [\sigma_1]\mathcal{J}) & \text{By inversion} \\
A_2' + \cdot \vdash [\sigma_2]\ (\textsc{combine2}\ (A_1')(*) \;\rhd\; [\sigma_1]\mathcal{J}) & \\
\cdot \vdash A_2' \leq A_2 \quad \text{and} \quad \vdash v_2 : A_2' & \text{By IH} \\
A_2' + \cdot \vdash \textsc{combine2}\ (A_1')(*) \;\rhd\; [\sigma_2][\sigma_1]\mathcal{J} & \text{By defn. of subst.} \\
A_2' + \cdot \vdash \textsc{combine2}\ (A_1')(*) \;\rhd\; [\sigma]\mathcal{J} & \text{By defn. of subst.}
\end{array}
$$

☞ $\mathcal{D}' :: \underbrace{(A_1' * A_2')}_{A'} + \cdot \vdash [\sigma]\mathcal{J}$ By inversion

(i) ☞ $\cdot \vdash \underbrace{A_1' * A_2'}_{A'} \leq \underbrace{A_1 * A_2}_{A}$ By rule $*$

$\cdot \vdash v_1 : A_1'$ Above (IH (ii), 1st subd.)

$\cdot \vdash v_2 : A_2'$ Above (IH (ii), 2nd subd.)

(ii) ☞ $\cdot \vdash (v_1, v_2) : A_1' * A_2'$ By $*$I

- **Case** as-p: $\mathcal{D} :: \dfrac{\cdot \vdash p_0 \downarrow A \;\; \triangleright \;\; (\text{AS } x \;\; \triangleright \;\; \mathcal{J})}{\cdot \vdash x \text{ as } p_0 \downarrow A \;\; \triangleright \;\; \mathcal{J}}$

  $p = x$ as p_0. By inversion on x as $p_0 \xrightarrow{\sigma} v$, we have $\sigma = \sigma_0, v/x$ and $p_0 \xrightarrow{\sigma_0} v$. By IH on the subderivation $\vdash p_0 \downarrow A \;\; \triangleright \;\; (\text{AS } x \;\; \triangleright \;\; \mathcal{J})$, we obtain a sub-subderivation $\mathcal{D}'' :: A' + \cdot \vdash [\sigma_0](\text{AS } x \;\; \triangleright \;\; \mathcal{J})$ such that (i), (ii) hold (showing (i), (ii)). Pushing $[\sigma_0]$ through and applying inversion, we obtain

  $$A' + (\cdot, x{:}A') \vdash [\sigma_0]\mathcal{J}$$

  Following the case for var-p, we obtain $A' + \cdot \vdash [\sigma_0, v/x]\mathcal{J}$.

- **Case** $\bot$-p: Here $A = \bot$ and $\cdot \vdash v : \bot$, which is impossible by Theorem 2.17. Thus the case cannot arise.

- **Case** $\vee$-p: $A = A_1 \vee A_2$.

$\cdot \vdash v : A_1 \vee A_2$ Given

$\cdot \vdash v : A_1$ or $\underbrace{\cdot \vdash v : A_2}_{\text{assume w.l.o.g.}}$ By Theorem 2.17

$\cdot \vdash p \downarrow A_2 \;\; \triangleright \;\; \mathcal{J}$ Subderivation

$\mathcal{D}' :: A_2' + \cdot \vdash [\sigma]\mathcal{J}$ and $\cdot \vdash A_2' \leq A_2$ By IH

$\cdot \vdash A_2 \leq A_1 \vee A_2$ By $\vee$R$_2$

(i) ☞ $\cdot \vdash A_2' \leq A_1 \vee A_2$ By Lemma 2.9 (transitivity)

(ii) ☞ $\cdot \vdash v : \underbrace{A_2'}_{A'}$ IH (ii)

- **Case** Σ-p: $A = \Sigma a{:}\gamma.\, A_0$.

$\cdot \vdash v : \Sigma a{:}\gamma.\, A_0$ Given

$\exists i.$ $\cdot \vdash i : \gamma$ and $\cdot \vdash v : [i/a]\, A_0$ By Theorem 2.17

$\cdot, a{:}\gamma \vdash p \downarrow A_0 \;\; \triangleright \;\; \mathcal{J}$ Subderivation

$\cdot \vdash p \downarrow [i/a]A_0 \;\; \triangleright \;\; \mathcal{J}$ By Lemma 4.8

$A_0' + \cdot \vdash [\sigma]\mathcal{J}$ and $\cdot \vdash A_0' \leq [i/a]A_0$ } By IH
(ii) ☞ (ii) holds for A_0'

$\cdot \vdash A_0 \leq \Sigma a{:}\gamma.\, A_0$ By ΣR

(i) ☞ $\cdot \vdash A_0' \leq \Sigma a{:}\gamma.\, A_0$ By Lemma 2.9 (transitivity)

- **Case** $\otimes$-p: Similar to the Σ-p case.

- **Case** $\wedge$-p: Broadly similar to the $(_,_)$-p case. (To show (i), use subtyping rules $\wedge L_1$, $\wedge L_2$, $\wedge R$ instead of rule $*$.)

$\vdash p \downarrow A_1 \ \rhd\ (\textsc{combine}\ (\wedge)(p \downarrow A_2)\ \rhd\ \mathcal{J})$ — Subderivation

$\vdash v : A_1 \wedge A_2$ — Given

$\vdash v : A_1$ and $\vdash v : A_2$ — By $\wedge E_1$ and $\wedge E_2$

$A_1' + \cdot \vdash \textsc{combine}\ (\wedge)(p \downarrow A_2)\ \rhd\ [\sigma]\mathcal{J}$ — By IH

$\cdot \vdash A_1' \le A_1$ and $\vdash v : A_1'$ — By IH

$\vdash p \downarrow A_2 \ \rhd\ (\textsc{combine2}\ (A_1')(\wedge)\ \rhd\ [\sigma]\mathcal{J})$ — By inversion

$A_2' + \cdot \vdash [\sigma]\ (\textsc{combine2}\ (A_1')(\wedge)\ \rhd\ [\sigma]\mathcal{J})$ — By IH

$\cdot \vdash A_2' \le A_2$ and $\vdash v : A_2'$ — By IH

$A_2' + \cdot \vdash \textsc{combine2}\ (A_1')(\wedge)\ \rhd\ [\sigma][\sigma]\mathcal{J}$ — By defn. of subst.

$A_2' + \cdot \vdash \textsc{combine2}\ (A_1')(\wedge)\ \rhd\ [\sigma]\mathcal{J}$ — By defn. of subst.

☞ $\mathcal{D}' :: \underbrace{(A_1' \wedge A_2')}_{A'} + \cdot \vdash [\sigma]\mathcal{J}$ — By inversion

(i) ☞ $\cdot \vdash \underbrace{A_1' \wedge A_2'}_{A'} \le \underbrace{A_1 \wedge A_2}_{A}$ — By $\wedge L_1$, $\wedge L_2$, $\wedge R$

$\cdot \vdash v : A_1'$ — Above (IH (ii), 1st subd.)

$\cdot \vdash v : A_2'$ — Above (IH (ii), 2nd subd.)

(ii) ☞ $\cdot \vdash v : A_1' \wedge A_2'$ — By $\wedge I$

- **Case** c-p: $p = c(p_0)$ and $A = \delta(i)$.

$c(p_0) \xrightarrow{\sigma} v$ — Given

$v = c(v_0)$ and $p_0 \xrightarrow{\sigma} v_0$ — By inversion

$\cdot \vdash c(v) : A$ — Given

$\cdot \vdash \delta(i) \le \delta(i)$ — By Lemma 2.9

$\cdot \not\models \bot$ — By Proposition 2.5

$\exists \delta', i', B_1.\quad \cdot \vdash \mathcal{S}(c) \uparrow B_1 \to \delta'(i')$ — By Lemma 2.18

$\cdot \vdash v_0 : B_1$ — By Lemma 2.18

$\cdot \vdash \delta'(i') \le \delta(i)$ — By Lemma 2.18

$\cdot ; c : \mathcal{S}(c); c(p_0) : \delta(i) \vdash \mathcal{J}$ — Subd.

$\cdot \vdash p_0 \downarrow B_1 \ \rhd\ (\textsc{con}\ c \downarrow \delta'(i')\ \rhd\ \mathcal{J})$ — By Lemma 4.9

By IH, $\mathcal{D}_0' :: B_1' + \cdot \vdash \textsc{con}\ c \downarrow \delta'(i')\ \rhd\ \mathcal{J}$ with
(i) $\cdot \vdash B_1' \le B_1$,
(ii) $\cdot \vdash v_0 : B_1'$.

By Lemma 4.10, $\mathcal{D}_0'$ has a subderivation $A' + \cdot \vdash \mathcal{J}$ such that $\cdot \vdash A' \le \delta'(i')$. We have $\cdot \vdash \delta'(i') \le \delta(i)$. By transitivity (Lemma 2.9) $\cdot \vdash A' \le \delta(i)$, satisfying obligation (i).

	$\cdot \vdash c : C_1 \to C_2$	Above (Lemma 4.10)
	$\cdot \vdash B_1' \leq C_1$	Above (Lemma 4.10)
	$\cdot \vdash v_0 : B_1'$	Above (IH (ii))
	$\cdot \vdash v_0 : C_1$	By sub
	$\cdot \vdash c(v_0) : C_2$	By δI
	$\cdot \vdash C_2 \leq A'$	Above (Lemma 4.10)
(ii) ☞	$\cdot \vdash c(v_0) : A'$	By sub

- **Case** {}-p: It is clear from Figure 4.1 that there exists no σ such that $\{\} \xrightarrow{\sigma} v$. Since $\{\} \xrightarrow{\sigma} v$ is given, the case cannot arise.
- **Case** $\sqcup$-p: By inversion on $p_1 \sqcup p_2 \xrightarrow{\sigma} v$, either $p_1 \xrightarrow{\sigma} v$ or $p_2 \xrightarrow{\sigma} v$. Applying the IH to the appropriate premise yields the result. □

Corollary 4.12 (Soundness of pattern checking). *If*

$$\cdot \vdash p \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e : C)$$

where $\cdot \vdash v : A$ *and* $p \xrightarrow{\sigma} v$ *then* $\cdot \vdash [\sigma]e : C$.

Proof. By Theorem 4.11 followed by inversion on $\cdot \vdash \textsc{forgettype} \;\rhd\; [\sigma]e : C$. □

Lemma 4.13. *If*

$$\cdot \vdash (p^* \downarrow A) \rhd ms : C$$

where $\cdot \vdash v : A$ *and* $v \in p^*$ *then there exists* e' *such that* **case** v **of** $ms \mapsto_M^+ e'$ *and* $\vdash e' : C$.

Proof. By induction on the first derivation. Only two rules could have concluded the judgment form.

- **Case** casearm:

$$\mathcal{D} :: \dfrac{FPV(p) \vdash e\ \mathsf{ok} \qquad \begin{array}{l}\cdot \vdash (p \cap p^*) \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e : C) \\ \cdot \vdash ((p^* - p) \downarrow A) \rhd ms' \;:\; C\end{array}}{\cdot \vdash (p^* \downarrow A) \rhd (p \Rightarrow e \mid ms') \;:\; C}$$

  We have $ms = (p \Rightarrow e \mid ms')$. Either $v \in p$ holds or it does not.

  - If it does not hold, we have both $v \in p^*$ (given) and $v \notin p$. By Property 4.3, $v \in p^* - p$. By IH on $\Gamma \vdash ((p^* - p) \downarrow A) \rhd ms' \;:\; C$, there exists e' such that **case** v **of** $ms' \mapsto_M^+ e'$ and $\vdash e' : C$. By rule ev-no-match, **case** v **of** $p \Rightarrow e \mid ms' \mapsto_M$ **case** v **of** ms'. It follows that **case** v **of** $p \Rightarrow e \mid ms' \mapsto_M^+ e'$.
  - If it does hold, we have both $v \in p^*$ and $v \in p$. By Property 4.4, $v \in p \cap p^*$. By $v \in p$, there exists σ_p such that $p \xrightarrow{\sigma_p} v$.
    We have a subderivation of $\cdot \vdash (p \cap p^*) \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e : C)$. By Lemma 4.8,

$$\cdot \vdash (p \cap p^*) \downarrow A \;\rhd\; (\textsc{forgettype} \;\rhd\; e : C)$$

By Theorem 4.11, $A' + \cdot \vdash \text{FORGETTYPE} \;\rhd\; [\sigma_p]e : C$ for some A'. By inversion, $\cdot \vdash [\sigma_p]e : C$. With $[\sigma_p]e$ as our e', we have $\cdot \vdash e' : C$.

By ev-match, $(\textbf{case}\ v\ \textbf{of}\ p \Rightarrow e \mid ms) \mapsto [\sigma_p]\, e$, which is e'.

- **Case** emptyms: $\mathcal{D} :: \dfrac{\cdot \vdash p^* \downarrow A \rhd \text{IMPOSSIBLE}}{\cdot \vdash (p^* \downarrow A) \rhd \cdot : C}$

  Following the reasoning in the second subcase above, with IMPOSSIBLE in place of FORGETTYPE $\rhd$ ..., we get

  $$A' + \cdot \vdash \text{IMPOSSIBLE}$$

  for some A'. The only rule that can possibly conclude $A' + \cdot \vdash \text{IMPOSSIBLE}$ is contra-$\mathcal{J}$, with premise $\cdot \models \bot$. However, $\cdot \not\models \bot$ by Property 2.5. We have a contradiction, so this case cannot arise. □

Now we can prove the extension of Theorem 2.21 to the richer pattern language.

Theorem 4.14 (Type Preservation and Progress (Full Pattern Language)). *(Restatement of Theorem 2.21.)*

If $\cdot \vdash e : C$ *then either*

(1) e value*, or*

(2) there exists e' *such that* $e \mapsto e'$ *and* $\vdash e' : C$.

Proof. Following Theorem 2.21, except in the δE case.

- **Case** δE: $\mathcal{D} :: \dfrac{\cdot \vdash e : A \qquad \cdot \vdash (_ \downarrow A) \rhd ms \;:\; C}{\cdot \vdash \textbf{case}\ e\ \textbf{of}\ ms : C}$

  By IH, either e value or there is some e' such that $e \mapsto e'$. For the second case, see the proof of Theorem 2.21. For the first case, we have e value. By Lemma 4.13, there exists e' such that $\textbf{case}\ e\ \textbf{of}\ ms \mapsto^+_M e'$. By rule ev-match$^+$, $\textbf{case}\ e\ \textbf{of}\ ms \mapsto_R [\sigma]e'$. By rule ev-context, $\textbf{case}\ e\ \textbf{of}\ ms \mapsto e'$. □

4.7 Limitations

We do not have a type-directed rule for every property type: there are no rules for most of the definite property types.

- The rule for $\top$, if included, would be

$$\frac{\top + \Gamma \vdash \mathcal{J}}{\Gamma \vdash p \downarrow \top \;\rhd\; \mathcal{J}}$$

which provides no information; wherever such a rule could be applied it would be at least as helpful to apply a pattern-directed rule.

- The rules for Π and $\supset$ would be

$$\frac{\Gamma, a{:}\gamma \vdash p \downarrow A \;\rhd\; (\textsc{univ}\ a \;\rhd\; \mathcal{J})}{\Gamma \vdash p \downarrow \Pi a{:}\gamma.\, A \;\rhd\; \mathcal{J}}\ \Pi\text{-p} \qquad \frac{(\Pi a{:}\gamma.\, A) + (\Gamma_1, \Pi a{:}\gamma.\, \Gamma_2) \vdash \mathcal{J}}{A + \Gamma_1, a{:}\gamma, \Gamma_2 \vdash \textsc{univ}\ a \;\rhd\; \mathcal{J}}$$

$$\frac{\Gamma, P \vdash p \downarrow A \;\rhd\; (\textsc{guard}\ P \;\rhd\; \mathcal{J})}{\Gamma \vdash p \downarrow P \supset A \;\rhd\; \mathcal{J}}\ \supset\text{-p} \qquad \frac{(P \supset A) + \Gamma_1, P \supset \Gamma_2 \vdash \mathcal{J}}{A + \Gamma_1, P, \Gamma_2 \vdash \textsc{guard}\ P \;\rhd\; \mathcal{J}}$$

where "$\Pi a{:}\gamma.\, \Gamma_2$" and "$P \supset \Gamma_2$" distribute into Γ_2: for example, $\Pi a{:}\gamma.\, (P, x{:}B) = (\forall a{:}\gamma.\ P), (\Pi a{:}\gamma.\, x{:}B)$. (Note the "$\forall a{:}\gamma.\ P$", a proposition form not otherwise required by the formalism, though it is required for constraints in the implementation.)

These rules are excluded because of a difficulty with quantifier alternation. Suppose $\mathcal{S}(c) = (\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, B) \to \delta(0)$. (Having a constructor with such a type might seem strange, but our system permits it.) When checking a pattern $c(p)$, we will check $\vdash p \downarrow \Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, B \;\rhd\; \mathcal{J}$. Its derivation will have the form

$$\dfrac{\dfrac{\dfrac{\dfrac{(\Pi a{:}\gamma.\, B') + \Gamma, \Pi a{:}\gamma.\, (b{:}\gamma, \Gamma_2) \vdash \mathcal{J}}{B' + \Gamma, a{:}\gamma, b{:}\gamma, \Gamma_2 \vdash \textsc{univ}\ a \;\rhd\; \mathcal{J}}}{\vdots}}{\Gamma, a{:}\gamma, b{:}\gamma \vdash p \downarrow B \;\rhd\; (\textsc{univ}\ a \;\rhd\; \mathcal{J})}}{\dfrac{\Gamma, a{:}\gamma \vdash p \downarrow \Sigma a{:}\gamma.\, B \;\rhd\; (\textsc{univ}\ a \;\rhd\; \mathcal{J})}{\Gamma \vdash p \downarrow \Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, B \;\rhd\; \mathcal{J}}\ \Pi\text{-p}}\ \Sigma\text{-p}$$

But what is $\Pi a{:}\gamma.\, (b{:}\gamma, \Gamma_2)$, exactly? It cannot be $\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, \Gamma_2$ (pushing the Σ inside in the same way as the Π), because the b will get duplicated in each assumption. For example, if $p = (x_1, x_2)$ and $B = B_1 * B_2$, we will get assumptions $x_1{:}\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, B_1, x_2{:}\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, B_2$. We no longer know that the b in B_1 and B_2 is the same.

Nor can it be $b{:}\gamma, \Pi a{:}\gamma.\, \Gamma_2$; then we would end up assuming $b{:}\gamma, x_1{:}\Pi a{:}\gamma.\, B_1, x_2{:}\Pi a{:}\gamma.\, B_2$, an obviously erroneous reversal of quantifier order.

There appears to be no easy way out. The core issue is that the proposition that naturally represents the typechecking problem, given the information gleaned from pattern checking, has the shape $\big(\forall a.\exists b.(p_1(a,b) \wedge p_2(a,b))\big) \Rightarrow q$, where p_1 is about x_1, p_2 is about x_2, and q is roughly "such and such a term is well typed". However, the judgment $\Gamma \vdash \ldots$ cannot have such a shape.

Fortunately, the constructor types that lead us into this issue do not seem to arise often. Types like $c : (\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, A_1 \to A_2) \to \delta(i)$ are no problem, since functions cannot be meaningfully pattern-matched: given a pattern $c(p)$ the p cannot break down the function and decompose $A_1 \to A_2$. In terms of the derivation tree above, nothing interesting can happen in the elided portion. Without the above rules, for $c(x)$ we will get the assumption $x : (\Pi a{:}\gamma.\, \Sigma b{:}\gamma.\, A_1 \to A_2)$, which is clearly the best possible.

4.8 Implementation

A direct implementation of the pattern checking system involves too much backtracking to be practical. A major factor is the first premise of rule CON-p, $\Gamma \vdash c : C_1 \rightarrow C_2$, in which a "component" $C_1 \rightarrow C_2$ of a constructor type is chosen. Removing redundant components turns out to be quite effective. Given a pattern $c(x)$ and value $c(v) : C_2$, a component $C_1' \rightarrow C_2$ of $\mathcal{S}(c)$ is redundant if there is a $C_1 \rightarrow C_2$ such that $C_1' \leq C_1$. At first glance this might seem backward: don't we want the smaller type C_1' that gives the best information about the domain? But either $C_1' \rightarrow C_2$ or $C_1 \rightarrow C_2$ might have been applied to conclude $c(v) : C_2$: we must consider both cases, checking the case arm under $x{:}C_1'$ and then under $x{:}C_1$. However, since C_1 is a supertype of C_1', if we succeed while assuming $x{:}C_1$ we will certainly succeed assuming $x{:}C_1'$. On the other hand, succeeding under the assumption $x{:}C_1'$ does not guarantee success assuming $x{:}C_1$. So we can safely ignore the $C_1' \rightarrow C_2$ component.

Thus, the implementation does not directly follow the logical system: the code corresponding to rule CON-p repeatedly 'invokes' $\overline{\Gamma} \vdash c : -$ to produce all the results C such that $\overline{\Gamma} \vdash c : C$, then strips out those having a domain that is "obviously" a subtype of another result's domain. A is obviously a subtype of B if:

- $A = \delta_A(i_A)$ and $B = \delta_B(i_B)$ and $\delta_A \preceq \delta_B$ and $i_A = i_B$ (literally); or
- A and B are products and are componentwise obvious subtypes; or
- $A = \Sigma a{:}\gamma.\, A'$ and $B = \Sigma b{:}\gamma.\, B'$ and A' is obviously a subtype of $[a/b]B'$.

Unlike real subtyping $\leq$, the "obviously" test cannot cause backtracking (and does not generate index constraints, since indices are compared literally, not with genuine index equality $\doteq$).

In practice, this strategy eliminates enough backtracking to cut the number of times a case arm with a moderately complicated pattern is checked by about an order of magnitude.

4.9 Related work

4.9.1 Pattern checking in unrefined type systems

In more traditional type systems such as that of Standard ML, basic concepts such as subtraction, intersection, and pattern matching itself ($p \xrightarrow{\sigma} v$) can be handled in similarly straightforward ways (though there seem to be few formal treatments of all of these concepts—for example, the SML Definition lacks a formal basis for redundancy and exhaustiveness checking, leading to an informal treatment of those topics [MTHM97, p. 28]). However, pattern checking itself is much simpler than in our system, primarily because information flows in only one direction during pattern checking: down toward the pattern's leaves; see the Definition, or the Harper-Stone semantics [HS97].

4.9.2 Davies' datasort refinement system

Davies' work on pattern checking is probably the most similar to ours: his system includes datasort refinements, intersection types, and even union types (the last introduced in a limited way specifically for pattern checking). He formulates pattern checking as a set of left rules whose assumptions

include pattern typings[8] of the form p:Z, where Z is a *pattern type*[9], which is essentially a pattern with *types instead of variables* at the leaves. For example, $c(A_1, c(A_2, A_3))$ is a valid pattern type. Every type is a valid pattern type (since x is a valid pattern, A must be a valid pattern type).

Davies' implementation supports all the pattern constructs of SML. However, his formal system does not support layered patterns x as p. Davies can justify this decision by arguing [Dav05a, pp. 238–9] that his implementation's handling of layered patterns is equivalent to typechecking after transforming the case arm by inserting a let-binding: x as $p \Rightarrow e$ can become $p \Rightarrow$ **let** $x = p$ **in** e since every well-formed pattern (with all "x as "s removed) is also a well-formed term. (His implementation does not actually do this transformation: it is merely a device to describe the semantics.) The idea of such a transformation is easy to understand, and is a reasonable bridge from the formal system without layered patterns. His argument works because pattern checking yields the same information about x that reconstructing the value and binding x to it would. We might like to make the same argument, which would allow us to drop layered patterns from the formal treatment and permit an arguably simpler formulation approximating Davies'. However, in our system, reconstructing the value does *not* yield the same information as pattern checking. This is due to the difference in the way refinements are specified: in Davies' system the user gives a regular tree grammar, from which inversion properties can be deduced and used in pattern checking; we assume, instead, a signature $\mathcal{S}$ of constructor types, which allows the specification of invaluable refinements (Section 7.4). For example, if we assume n : red, then the following is ill typed in our system:

```
case n of
   S(x as S(Z)) ⇒ (x : blue)
|  ...
```

because it attempts to use x at type blue, and the only thing the system can deduce from n : red and $n = S(\dots)$, according to the signature $\mathcal{S}(S) = (\mathsf{red} \to \mathsf{red}) \wedge (\mathsf{blue} \to \mathsf{blue})$, is that $\cdots$: red, i.e. x : red. Thus, pattern checking results in checking the arm with the assumption x : red. However, the values inhabiting red and blue are identical; if we transformed the program to

```
case n of
   S(S(Z)) ⇒ let x=S(Z) in (x : blue) end
|  ...
```

it would typecheck[10]!

[8]Where we say "type", Davies says "sort"; the reader can substitute appropriately. We likewise adapt Davies' notation to ours wherever possible in this discussion.

[9]That is, Davies' "pattern sort".

[10]Actually it would not, since $S(Z)$ is not a synthesizing form, but we could then speak of substituting $S(Z)$ for x in the case arm; $(S(Z) : \mathsf{blue})$ would typecheck, proving the point.

Chapter 5

A let-normal type system

5.1 Introduction

Previous chapters developed a rich system in which typechecking of indefinite property types, such as union and existential types, is based on a *tridirectional rule* that decomposes its subject term according to some evaluation context $\mathcal{E}$:

$$\frac{\Gamma; \Delta_1 \vdash e' \uparrow A \qquad \Gamma; \Delta_2, \bar{\mathbf{x}}{:}A \vdash \mathcal{E}[\bar{\mathbf{x}}] \downarrow C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{E}[e'] \downarrow C}\ \mathsf{direct}\mathbb{L}$$

This rule gives e' a (linear) name $\bar{\mathbf{x}}$, so that left rules can eliminate union and existential types appearing in A. While the notion of evaluation context is purely syntactic, this rule is not syntax-directed in the usual sense, as many terms have more than one decomposition into some $\mathcal{E}[e']$ where the subterm[1] e' can synthesize a type. For example, $f(x, y)$ has four decompositions: $\mathcal{E} = []$, $\mathcal{E} = [](x, y)$, $\mathcal{E} = f([], y)$, and $\mathcal{E} = f(x, [])$. Thus, a straightforward implementation of the system of Chapter 3 would require far too much backtracking. Compounded with backtracking due to intersection and union types (e.g. if $f : (A_1 \rightarrow A_2) \wedge (B_1 \rightarrow B_2)$ we may have to try both $f : A_1 \rightarrow A_2$ and $f : B_1 \rightarrow B_2$), such a straightforward implementation is clearly impractical.

In this chapter, we reformulate the earlier system (summarized in Section 5.2) to work on terms in a particular *let-normal form*. The let-normal transformation (Section 5.3) drastically constrains the decomposition by sequentializing terms, forcing typechecking to proceed left to right (with an interesting exception). Our soundness and completeness results in Sections 5.6 and 5.7 guarantee that the let-normal version of a program e is well typed under the let-normal version of the type system if and only if e is well typed under the tridirectional system.

The details of the transformation may be of interest to designers of advanced type systems, whether their need for a sequentialized form arises from typechecking per se (as in this case) or from a concern for compiler efficiency.

A warning to the reader: This chapter is quite long. The let-normal translation itself is not complicated, though the motivation for our particular formulation is somewhat involved. The real pain is in the proof of completeness.

[1] In this discussion, "subterm" means "subterm in synthesizing syntactic form"; checking forms such as (e_1, e_2) cannot be named by $\mathsf{direct}\mathbb{L}$ because they can never synthesize a type.

5.2 Tridirectional typechecking

When this chapter mentions "the tridirectional system", it means the *left* tridirectional system (or "left rule system"). The "simple tridirectional system" in the first part of Chapter 3 is not needed here; the soundness and completeness proofs will be with respect to the left tridirectional system.

5.2.1 Evaluation contexts do not strictly determine order

Rule $\mathsf{direct}\mathbb{L}$'s use of an evaluation context might give the impression that typechecking simply proceeds in the order in which terms are actually evaluated. However, this is not the case. The subject of $\mathsf{direct}\mathbb{L}$ is $\mathcal{E}[e']$ where e' synthesizes a type, so certainly e' must be in *an* evaluation position, but there may be several such positions. Even a term as simple as $f\ (x\ y)$ has 5 subterms in evaluation position, each corresponding to a different evaluation context $\mathcal{E}$:

$$\begin{array}{lll@{\qquad}lll} \mathcal{E} = [\,]\ (x\ y) & \text{and} & e' = f & \mathcal{E} = f\ [\,] & \text{and} & e' = (x\ y) \\ \mathcal{E} = f\ ([\,]\ y) & \text{and} & e' = x & \mathcal{E} = [\,] & \text{and} & e' = f\ (x\ y) \\ \mathcal{E} = f\ (x\ [\,]) & \text{and} & e' = y & & & \end{array}$$

In fact, we may need to repeatedly apply $\mathsf{direct}\mathbb{L}$ to the same subject term with different choices of $\mathcal{E}$! For example, we might use $\mathcal{E} = [\,]\ (x\ y)$ to name an f of union type, introducing $\bar{\mathbf{f}}{:}A \vee B$ into the context; then, case-analyze $A \vee B$ with $\vee$E; finally, choose $\mathcal{E} = \bar{\mathbf{f}}\ ([\,]\ y)$ to name x (also of union type). Thus we are faced not with a choice over decompositions, but over *sequences* of decompositions.

It is an essential fact that typechecking cannot always proceed left to right. Consider

$$(\mathit{map}\ f)\ (\mathit{filter}\ xs)$$

where $\mathit{map} : (\mathsf{int} \to \mathsf{int}) \to \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a)$ and $\mathit{filter} : \Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \Sigma b'{:}\mathcal{N}.\, \mathsf{list}(b')$. (For simplicity, we omit the functional argument to *filter*.)

The term $(\mathit{map}\ f)$ synthesizes a Π type, which must be eliminated—by instantiating a to some index—so that rule $\to$E can be applied to $(\mathit{map}\ f)\ (\mathit{filter}\ xs)$. However, the proper instantiation of a is b' (from the type of *filter*), which is unknown since the $(\mathit{filter}\ xs)$ subterm has not yet been visited.

How do we "jump over" $(\mathit{map}\ f)$ to type $(\mathit{filter}\ xs)$ first so we can get a b' to plug in for a? First apply $\mathsf{direct}\mathbb{L}$ with evaluation context $\mathcal{E} = [\,]\ (\mathit{filter}\ xs)$, giving $(\mathit{map}\ f)$ the name $\bar{\mathbf{x}}$; second, apply $\mathsf{direct}\mathbb{L}$ with context $\mathcal{E} = \bar{\mathbf{x}}\ [\,]$, synthesizing $\Sigma b'{:}\mathcal{N}.\, \mathsf{list}(b')$ for $(\mathit{filter}\ xs)$ and introducing the assumption $\bar{\mathbf{y}} : \Sigma b'{:}\mathcal{N}.\, \mathsf{list}(b')$. Now apply the left existential rule $\Sigma\mathbb{L}$ to unpack the Σ, introducing the assumption $b' : \mathcal{N}$ while changing $\bar{\mathbf{y}} : \Sigma b'{:}\mathcal{N}.\, \mathsf{list}(b')$ to $\bar{\mathbf{y}} : \mathsf{list}(b')$ in the context Δ. Since b' is now known, a can be instantiated to it.

On a purely theoretical level, the tridirectional system is satisfactory, but the nondeterminism is excessive. Xi approached (very nearly) the same problem by transforming the program so the term of Σ type appears early enough to instantiate the Π. A standard let-normal translation $|e|$ [Xi98, p. 86], where

$$|e_1\, e_2| = \textbf{let}\ x_1 = |e_1|\ \textbf{in}\ \textbf{let}\ x_2 = |e_2|\ \textbf{in}\ x_1 x_2$$

suffices for the examples above. (In Xi's system, existential variables are unpacked where a term of existential type is let-bound: b' is unpacked at the binding of x_2, which appears before the

application $x_1 x_2$ at which a must be instantiated.) Unfortunately, the translation interacts unpleasantly with bidirectionality: terms such as *map* $(\lambda x.\, e)$, in which $(\lambda x.\, e)$ must be checked, no longer typecheck because the λ becomes the right hand side of a **let**, in

$$\textbf{let}\ x_1 = \textit{map}\ \textbf{in let}\ x_2 = \lambda x.\, e\ \textbf{in}\ x_1 x_2$$

and let-bound expressions must synthesize a type, but $\lambda x.\, e$ does not. Thus typability is lost in translation; typechecking becomes incomplete in the sense that certain programs that are well typed before translation are not well typed after translation.

Xi ameliorated this incompleteness by treating $e_1\, v_2$ as a special case [Xi98, p. 139]:

$$|e_1\, v_2| \;=\; \textbf{let}\ x_1 = |e_1|\ \textbf{in}\ x_1\, v_2$$

Now v_2 (which is $\lambda x.\, e$ in the above example) is in a checking position. This is adequate for non-synthesizing values, but terms such as *map* $(\textbf{case}\ z\ \textbf{of}\ \ldots)$, where a non-synthesizing non-value is in checking position, remain untypable. It is not clear why Xi did not add corresponding special cases for **case** and other non-synthesizing non-values, e.g.

$$|e_1\, (\textbf{case}\ e\ \textbf{of}\ ms)| \;=\; \textbf{let}\ x_1 = |e_1|\ \textbf{in}\ x_1\, |\textbf{case}\ e\ \textbf{of}\ ms|$$

Another region of incompleteness is evident in terms such as $f\ (\textbf{case}\ x\ \textbf{of}\ ms)$. Suppose x synthesizes an existential that must be unpacked to eliminate a Π quantifier on the type of f. Since x's scope—and thus the scope of its existential—is entirely within the **let** created for the **case**, typechecking fails.

$$\begin{aligned} |f\,(\textbf{case}\ x\ \textbf{of}\ ms)| &= \textbf{let}\ f_1 = f\ \textbf{in let}\ x_0 = |\textbf{case}\ x\ \textbf{of}\ ms|\ \textbf{in}\ f_1\ x_0 \\ &= \textbf{let}\ f_1 = f\ \textbf{in let}\ x_0 = \big(\textbf{let}\ x_1 = x\ \textbf{in case}\ x_1\ \textbf{of}\ |ms|\big)\ \textbf{in}\ f_1\ x_0 \end{aligned}$$

It could be argued that the cases in which Xi's translation fails are rare in practice. However, that may only increase confusion when such a case is encountered.

We follow Xi's general approach of sequentializing the program before typechecking, but no programs are lost in our translation.

5.2.2 Approaching the problem

Do we need all the freedom that direct$\mathbb{L}$ provides? No. At the very least, if we do not need to name a subterm, naming it anyway does no harm. But this only slightly reduces the nondeterminism. Clearly, a strategy of in-order traversal is sound (we can choose to apply direct$\mathbb{L}$ from left to right if we like). It is tempting to think it is complete. This conjecture (which we believed for some time) holds for many programs, but fails for a certain class of annotated terms. We will explain why as we present the general mechanism for enforcing a strategy of left-to-right traversal except for certain annotated terms.

5.3 Let-normal typechecking

This section explains and motivates our particular formulation of let-normalization. We briefly discuss previous work on let-normal forms, then explain the ideas behind our variant, including why we need to prove a *principal synthesis of values* property. Because the most universal form of principality does not hold for a few terms, we introduce *slack bindings*.

Traditional let-normal or A-normal transformations [Mog88, FSDF93]

1. explicitly sequence the computation, and

2. name the result of each intermediate computation.

(One may compare these to continuation-passing style (CPS) [Rey93], which in addition

3. introduces named continuations.

Thus let-normal form is sometimes called *two-thirds CPS*.) Many compilers for functional languages, to facilitate optimizations, translate programs to some kind of let-normal form; see, for instance, Tarditi et al. [TMC+96], Tolmach and Oliva [TO98], Reppy [Rep01], and MacKenzie and Wolverson [MW03].

Our variant of let-normal form sequentializes the computation in much the usual way. However, it does not only name intermediate *computations*, but *values* as well. In our let-normal type system, $\mathsf{direct}\mathbb{L}$ is replaced by a rule let that can *only* be applied to **let**:[2]

$$\left[\;\dfrac{\begin{array}{l} e' \text{ not a linear var} \\ \Gamma;\Delta_1 \vdash e' \uparrow A \qquad \Gamma;\Delta_2,\overline{\mathbf{x}}{:}A \vdash \mathcal{E}[\overline{\mathbf{x}}] \downarrow C \end{array}}{\Gamma;\Delta_1,\Delta_2 \vdash \mathcal{E}[e'] \downarrow C}\;\mathsf{direct}\mathbb{L}\;\right]$$

$$\dfrac{\Gamma;\Delta_1 \vdash e' \uparrow A \qquad \Gamma;\Delta_2,\overline{\mathbf{x}}{:}A \vdash \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}{\Gamma;\Delta_1,\Delta_2 \vdash \mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}\;\mathsf{let}$$

Our **let** is a syntactic marker with no computational character. In contrast to let-normal translations for compilation purposes, there is no evaluation step (reduction) corresponding to a **let**. In fact we do not even give a dynamic semantics for terms with **let**s. Such a semantics would not be difficult; it is simply not useful here, since our let-normal form is not intended for use in compilation. If we insist on knowing what a let-normal term e "means", we can use the operational semantics in Chapter 2 on the term's reverse translation $\hookleftarrow\!(e)$ (defined below).

Instead of making explicit the order of computation, our let-normal form makes explicit the order of *typechecking*—specifically, the order in which $\mathsf{direct}\mathbb{L}$ names subterms in evaluation position. Thus, to be complete with respect to the tridirectional system, the transformation must create a **let** for every subterm in synthesizing form: if an (untranslated) program contains a subterm e' in synthesizing form, it might be possible to name e' with $\mathsf{direct}\mathbb{L}$, so the let-normal translation must contain a **let** binding e'.[3] Otherwise, necessary opportunities to apply left rules such as $\vee\mathbb{L}$ may be

[2] $\mathcal{Q}$ is an elongated evaluation context, defined below.

[3] The subterm itself may change in the translation, so this is not strictly accurate.

lost. Even variables x must be named, since they synthesize a type and so can be named in direct$\mathbb{L}$. On the other hand, terms in checking form cannot synthesize, so they are not named. Again, these aspects are motivated by a desire to sequentialize the term according to (an "aggressive left-to-right" strategy for) application of direct$\mathbb{L}$.

Another consequence of our let-normal form following typechecking, not evaluation, is that $\mathbf{let}\ \overline{\mathbf{x}} = v_1\ \mathbf{in}\ v_2$ is considered a value—after all, the original term $[v_1/\overline{\mathbf{x}}]\, v_2$ is a value, and if the let-normal transformation made a value into a non-value, we could not apply value-restricted typing rules such as $\wedge$I, leading to incompleteness.

We define the translation via a judgment $e \hookrightarrow L + e'$, read "e translates to a sequence of let-bindings L with body e'". For example, the translation of $f\ (x\ y)$, which names every synthesizing subterm, is

$$\mathbf{let}\ \overline{\mathbf{f}} = f\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{x}} = x\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{y}} = y\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{a}} = \overline{\mathbf{f}}\ \overline{\mathbf{z}}\ \mathbf{in}\ \overline{\mathbf{a}}$$

This is expressed by the judgment

$$f\ (x\ y) \hookrightarrow \overline{\mathbf{f}} = f, \overline{\mathbf{x}} = x, \overline{\mathbf{y}} = y, \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}, \overline{\mathbf{a}} = \overline{\mathbf{f}}\ \overline{\mathbf{z}} + \overline{\mathbf{a}}$$

The definition is given in Figure 5.2. Note that $L + e'$ is not a term; $+$ is punctuation in the translation judgment. We write $L\ \mathbf{in}\ e'$ as shorthand for the obvious expansion: when one sees $e \hookrightarrow L + e'$ one should think $L\ \mathbf{in}\ e'$. The distinction between the two notations arises from the multiple ways in which a term can be decomposed into a pair of a sequence of bindings and a "body" term. For example, $\mathbf{let}\ \overline{\mathbf{x}}_1 = e_1\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{x}}_2 = e_2\ \mathbf{in}\ e_3$ has three possible decompositions, depending on how many bindings one chooses to include in L:

$$\begin{aligned} \cdot\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{x}}_1 = e_1\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{x}}_2 = e_2\ \mathbf{in}\ e_3 &= (\overline{\mathbf{x}}_1 = e_1)\ \mathbf{in}\ \mathbf{let}\ \overline{\mathbf{x}}_2 = e_2\ \mathbf{in}\ e_3 \\ &= (\overline{\mathbf{x}}_1 = e_1, \overline{\mathbf{x}}_2 = e_2)\ \mathbf{in}\ e_3 \end{aligned}$$

We call the last decomposition *maximal*: it has the maximum number of bindings (and the smallest 'body'), which is the case exactly when the body is not itself a **let**. If $e \hookrightarrow L + e'$, it is the case that $L\ \mathbf{in}\ e'$ is maximal (Proposition 5.16); but this is because we distinguish the notations and use $+$ exclusively in the definition of $\hookrightarrow$. If we did not, the definition would allow one to reinterpret the right hand side of a premise as a non-maximal decomposition. Fortunately, this rather tedious distinction really matters only in Lemma 5.72; generally, one can read $L + e'$ the same as $L\ \mathbf{in}\ e'$.

To precisely model an aggressive left-to-right strategy of direct$\mathbb{L}$-application, we must translate $e \hookrightarrow L + e'$ with L binding exactly those subterms that could be in evaluation position (after zero or more "preliminary" applications of direct$\mathbb{L}$). We draw a syntactic distinction between

The háček, ˇ, above the e is shaped like a 'v', for 'value'.

- *pre-values* $\check{e}$: terms that are either values (such as x) or that can "become" values in the derivation via direct$\mathbb{L}$ (such as $x\ y$, which "becomes" $\overline{\mathbf{z}}$, which is a value), and
- *anti-values* $\widehat{e}$: terms such as $\mathbf{case}\ e_0\ \mathbf{of}\ ms$ that are not values and cannot become values.

direct$\mathbb{L}$ can replace any synthesizing subterm with a linear variable, so the prevalues must include both the values and the synthesizing forms. Also, terms such as $c(x\ y)$ must be prevalues: $c(x\ y)$ is neither a value nor a synthesizing form, but by applying direct$\mathbb{L}$ with $E = c([\,])$ we have a subderivation typing $c(\overline{\mathbf{z}})$, which *is* a value since $c(e)$ value iff e value. Likewise, (e_1, e_2) is a prevalue

if *both* e_1 and e_2 are—if either one is a **case**, say, there is no way to turn the whole pair into a value. This leads to the following grammar, with values x, $\overline{\mathbf{x}}$, $\lambda x.\, e$, and $()$; synthesizing forms $(e : As)$, $e_1 e_2$, u, $\mathbf{fst}(e)$, and $\mathbf{snd}(e)$; and checking forms that can become values if all their subterms can: (e_1, e_2) and $c(e)$.

$$\text{Pre-values} \quad \check{e} \quad ::= \quad x \mid \overline{\mathbf{x}} \mid (e : As) \mid \lambda x.\, e \mid e_1 e_2 \mid u \mid (\check{e}_1, \check{e}_2) \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid () \mid c(\check{e})$$

We write e prevalue to mean that a given e is a pre-value, and e antivalue to mean e is an anti-value. Every e is either a pre- or anti-value, leading us to the following grammar for the anti-values.

$$\text{Anti-values} \quad \widehat{e} \quad ::= \quad \mathbf{case}\; e\; \mathbf{of}\; ms \mid \mathbf{fix}\; u.\, e \mid (\widehat{e}_1, e_2) \mid (e_1, \widehat{e}_2) \mid c(\widehat{e})$$

It is easy to verify that for any e we have either e antivalue or e prevalue, but not both.

The distinction comes into play in the translation only for terms with sequences of immediate subterms such that at least two subterms in the sequence may be in evaluation position. Only application $e_1 e_2$ and pairing (e_1, e_2) have this property; other terms, such as **case** e_0 **of** ms have subterms in sequence but only one of them (e_0) is in evaluation position, while $\lambda x.\, e$ and **fix** $u.\, e$ have no subterms in evaluation position at all.

A telling example is

$$(\mathbf{case}\; x\; \mathbf{of}\; c(y) \Rightarrow e, \omega\; x)$$

where $\omega : \top \to \bot$. In the left tridirectional system, the only subterm in evaluation position is x, and applying direct$\mathbb{L}$ to replace it with $\overline{\mathbf{x}}$ does not bring any other subterms into evaluation position. In particular, $\omega\; x$ is not in evaluation position, so however we translate the term, we must not bind $\omega\; x$ outside the pair; if we did, we would add $\overline{\mathbf{z}}{:}\bot$ to the context and could apply rule $\bot\mathbb{L}$ to declare the entire pair well typed without examining e! If e is $()\;()$, this is actually unsound. On the other hand, in the term

$$(f\; g, \omega\; x)$$

the left tridirectional system *can* bind $\omega\; x$ before checking the pair, by applying direct$\mathbb{L}$ with $E = ([\,], \omega\; x)$ to yield a subject $(\overline{\mathbf{x}}, \omega\; x)$ in which $\omega\; x$ is in evaluation position.

The difference is that **case** x **of** $c(y) \Rightarrow e$ is an anti-value, while $f\; g$ is a pre-value. Therefore, given a pair (e_1, e_2), if e_1 is some anti-value $\widehat{e}_1$, the translation places the bindings for subterms of e_2 (e.g. $\overline{\mathbf{z}} = \omega\; x$ above) *inside* the second component. On the other hand, if e_1 is a pre-value $\check{e}_1$, the translation puts the bindings for subterms of e_2 *outside* the pair.

$$\frac{\widehat{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{(\widehat{e}_1, e_2) \hookrightarrow L_1 + (e_1', L_2\; \mathbf{in}\; e_2')} \qquad \frac{\check{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{(\check{e}_1, e_2) \hookrightarrow L_1, L_2 + (e_1', e_2')}$$

We define *elongated evaluation contexts* $\mathcal{Q}$, which—unlike ordinary evaluation contexts $\mathcal{E}$—can skip over pre-values. Elongated evaluation position is a sort of transitive closure of evaluation position: if, by repeatedly replacing prevalues in evaluation position with values, some subterm is then in evaluation position, that subterm is in elongated evaluation position. This corresponds to a sequence of direct$\mathbb{L}$-applications: subterms in evaluation position are replaced with linear variables, which are values. For example, z is not in evaluation position in $(x\; y)\; z$, but applying direct$\mathbb{L}$ with $\mathcal{E} = [\,]\; z$ yields a subderivation with subject $\overline{\mathbf{x}}\; z$, in which z *is* in evaluation position. A $\mathcal{Q}$ is thus a

Anti-values	$\widehat{e}$	::=	$\textbf{case } e \textbf{ of } ms \mid \textbf{fix } u.\, e \mid (\widehat{e}_1, e_2) \mid (e_1, \widehat{e}_2) \mid c(\widehat{e})$
Pre-values	$\check{e}$	::=	$x \mid \overline{\mathbf{x}} \mid (e : As) \mid \lambda x.\, e \mid e_1 e_2 \mid u \mid (\check{e}_1, \check{e}_2) \mid \textbf{fst}(e) \mid \textbf{snd}(e) \mid () \mid c(\check{e})$
Elongated evaluation contexts	$\mathcal{Q}$	::=	$[\,] \mid \mathcal{Q}e \mid \check{e}\mathcal{Q} \mid (\mathcal{Q}, e) \mid (\check{e}, \mathcal{Q}) \mid \textbf{fst}(\mathcal{Q}) \mid \textbf{snd}(\mathcal{Q}) \mid c(\mathcal{Q}) \mid \textbf{case } \mathcal{Q} \textbf{ of } ms \mid (\mathcal{Q} : As)$ $\mid \textbf{let } \overline{\mathbf{x}} = \mathcal{Q} \textbf{ in } e \mid \textbf{let } \overline{\mathbf{x}} = \check{e} \textbf{ in } \mathcal{Q}$ $\mid \textbf{let } {\sim}\overline{\mathbf{x}} = \mathcal{Q} \textbf{ in } e \mid \textbf{let } {\sim}\overline{\mathbf{x}} = v \textbf{ in } \mathcal{Q}$
Terms	e	::=	$\ldots \mid \textbf{let } \overline{\mathbf{x}} = e_1 \textbf{ in } \mathcal{Q}[\overline{\mathbf{x}}] \mid \textbf{let } {\sim}\overline{\mathbf{x}} = v_1 \textbf{ in } \mathcal{Q}[\overline{\mathbf{x}}]$
Values	v	::=	$\ldots \mid \overline{\mathbf{x}} \mid \textbf{let } \overline{\mathbf{x}} = v_1 \textbf{ in } v_2 \mid \textbf{let } {\sim}\overline{\mathbf{x}} = v_1 \textbf{ in } v_2$
Eval. contexts	$\mathcal{E}$	::=	$\ldots \mid \textbf{let } \overline{\mathbf{x}} = \mathcal{E} \textbf{ in } e \mid \textbf{let } \overline{\mathbf{x}} = v \textbf{ in } \mathcal{E} \mid \textbf{let } {\sim}\overline{\mathbf{x}} = \mathcal{E} \textbf{ in } e \mid \textbf{let } {\sim}\overline{\mathbf{x}} = v \textbf{ in } \mathcal{E}$
Sequences of bindings	L	::=	$\cdot \mid L, (\overline{\mathbf{x}} = e) \mid L, ({\sim}\overline{\mathbf{x}} = v)$

Figure 5.1: Syntax of terms and contexts in the let-normal type system

path that can skip prevalues: if every intervening subterm is a pre-value (equivalently, if there is no intervening anti-value), the hole is in elongated evaluation position.

A subterm in elongated evaluation position is said to be *viable*. The grammar for let-normal terms enforces the invariant that let-bound variables are viable in their scopes: the body e_2 of $\textbf{let } \overline{\mathbf{x}} = e_1 \textbf{ in } e_2$ must have the form $\mathcal{Q}[\overline{\mathbf{x}}]$.

Definition 5.1 (Viable)**.** A term e' is a *viable subterm of* e iff $\mathcal{Q}[e'] = e$ for some $\mathcal{Q}$.

5.3.1 Principal synthesis of values

One of the key steps in the completeness proof is the movement of let-bindings outward. To prove that such movements preserve typing, we show that principal types [Hin69] exist in certain cases. To see why, consider the judgment

$$x : (A_1 \to B) \wedge (A_2 \to B), y : A_1 \vee A_2; \cdot \vdash x\, y \downarrow B$$

To derive this judgment in the left tridirectional system, we start by using $\mathsf{direct}\mathbb{L}$ with $\mathcal{E} = x\,[\,]$ to name y as a new linear variable $\overline{\mathbf{y}}{:}A_1 \vee A_2$. Then we use $\vee\mathbb{L}$ to decompose the union; we must now derive

$$x : (A_1 \to B) \wedge (A_2 \to B), \ldots; \overline{\mathbf{y}}{:}A_1 \vdash x\, \overline{\mathbf{y}} \downarrow B \quad \text{and} \quad x : (A_1 \to B) \wedge (A_2 \to B), \ldots; \overline{\mathbf{y}}{:}A_2 \vdash x\, \overline{\mathbf{y}} \downarrow B$$

Here, the scope of $\overline{\mathbf{y}}$ is $x\, \overline{\mathbf{y}}$, and we synthesize a type for x *twice*, once in each branch:

$$\dfrac{\ldots, y : A_1 \vee A_2; \cdot \vdash y \uparrow A_1 \vee A_2 \qquad \dfrac{\dfrac{\ldots; \cdot \vdash x \uparrow A_1 \to B \quad \vdots}{\ldots; \overline{\mathbf{y}}{:}A_1 \vdash x\, \overline{\mathbf{y}} \downarrow B}{\to}\mathrm{E} \qquad \dfrac{\ldots; \cdot \vdash x \uparrow A_2 \to B \quad \vdots}{\ldots; \overline{\mathbf{y}}{:}A_2 \vdash x\, \overline{\mathbf{y}} \downarrow B}{\to}\mathrm{E}}{\ldots; \overline{\mathbf{y}} : A_1 \vee A_2 \vdash x\, \overline{\mathbf{y}} \downarrow B}\vee\mathbb{L}}{x : (A_1 \to B) \wedge (A_2 \to B), y : A_1 \vee A_2; \cdot \vdash x\, y \downarrow B}\mathsf{direct}\mathbb{L}$$

$\boxed{e \hookrightarrow L + e'}$ read "e translates to bindings L with result e'"

$$\frac{}{x \hookrightarrow (\overline{\mathbf{x}} = x) + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e'}{\lambda x.\, e \hookrightarrow \cdot + \lambda x.\, (L \ \mathbf{in}\ e')}$$

$$\frac{\widehat{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{\widehat{e}_1 e_2 \hookrightarrow L_1, \overline{\mathbf{x}} = e_1'(L_2 \ \mathbf{in}\ e_2') + \overline{\mathbf{x}}} \qquad \frac{\check{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{\check{e}_1 e_2 \hookrightarrow L_1, L_2, \overline{\mathbf{x}} = e_1' e_2' + \overline{\mathbf{x}}}$$

$$\frac{}{u \hookrightarrow (\overline{\mathbf{x}} = u) + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e'}{\mathbf{fix}\ u.\, e \hookrightarrow \cdot + \mathbf{fix}\ u.\, (L \ \mathbf{in}\ e')}$$

$$\frac{e \hookrightarrow L + e' \qquad e \text{ not a value}}{(e : As) \hookrightarrow L, \overline{\mathbf{x}} = (e' : As) + \overline{\mathbf{x}}} \qquad \frac{v \hookrightarrow L + e'}{(v : As) \hookrightarrow L, \sim\overline{\mathbf{x}} = (e' : As) + \overline{\mathbf{x}}}$$

$$\frac{}{() \hookrightarrow \cdot + ()} \qquad \frac{\widehat{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{(\widehat{e}_1, e_2) \hookrightarrow L_1 + (e_1', L_2 \ \mathbf{in}\ e_2')} \qquad \frac{\check{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{(\check{e}_1, e_2) \hookrightarrow L_1, L_2 + (e_1', e_2')}$$

$$\frac{e \hookrightarrow L + e'}{\mathbf{fst}(e) \hookrightarrow L, \overline{\mathbf{x}} = \mathbf{fst}(e') + \overline{\mathbf{x}}} \qquad \frac{e \hookrightarrow L + e'}{\mathbf{snd}(e) \hookrightarrow L, \overline{\mathbf{x}} = \mathbf{snd}(e') + \overline{\mathbf{x}}}$$

$$\frac{e \hookrightarrow L + e'}{c(e) \hookrightarrow L + c(e')} \qquad \frac{e \hookrightarrow L + e' \qquad ms \hookrightarrow ms'}{\mathbf{case}\ e\ \mathbf{of}\ ms \hookrightarrow L + \mathbf{case}\ e'\ \mathbf{of}\ ms'}$$

$$\frac{}{\overline{\mathbf{x}} \hookrightarrow \cdot + \overline{\mathbf{x}}}$$

$\boxed{ms \hookrightarrow ms'}$ read "matches ms translate to ms'"

$$\frac{}{\cdot \hookrightarrow \cdot} \qquad \frac{e \hookrightarrow L + e' \qquad ms \hookrightarrow ms'}{(c(x) \Rightarrow e \mid ms) \hookrightarrow (c(x) \Rightarrow L \ \mathbf{in}\ e') \mid ms'}$$

Figure 5.2: Let-normal translation

However, when checking the translated term

$$\textbf{let}\ \overline{\mathbf{x}} = x\ \textbf{in let}\ \overline{\mathbf{y}} = y\ \textbf{in}\ \underbrace{\textbf{let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}}}$$

against B, we need to first name x as $\overline{\mathbf{x}}$, then y as $\overline{\mathbf{y}}$, then use $\vee\mathbb{L}$ to decompose the union $\overline{\mathbf{y}}{:}A_1 \vee A_2$ with subject $\textbf{let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}}$.

$$\frac{\ldots;\cdot \vdash x \uparrow (A_1{\to}B) \wedge (A_2{\to}B) \qquad \ldots;\overline{\mathbf{x}}{:}(A_1{\to}B) \wedge (A_2{\to}B) \vdash \textbf{let}\ \overline{\mathbf{y}} = y\ \textbf{in let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}} \downarrow B}{x : (A_1{\to}B) \wedge (A_2{\to}B), y : A_1 \vee A_2; \cdot \vdash \textbf{let}\ \overline{\mathbf{x}} = x\ \textbf{in let}\ \overline{\mathbf{y}} = y\ \textbf{in let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}} \downarrow B}\ \mathsf{let}$$

But we only get to synthesize a type for x *once* (at the point highlighted above), so we must take care when using let to name x; if we choose to synthesize $x \uparrow A_1{\to}B$ in let, we will be unable to derive $\overline{\mathbf{x}}{:}A_1{\to}B, \overline{\mathbf{y}}{:}A_2 \vdash \textbf{let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}} \downarrow B$ while if we choose to synthesize $x \uparrow A_2{\to}B$ we will be unable to derive $\overline{\mathbf{x}}{:}A_2{\to}B, \overline{\mathbf{y}}{:}A_1 \vdash \textbf{let}\ \overline{\mathbf{z}} = \overline{\mathbf{x}}\ \overline{\mathbf{y}}\ \textbf{in}\ \overline{\mathbf{z}} \downarrow B$. The only choice that works is $\Gamma(x) = (A_1{\to}B) \wedge (A_2{\to}B)$, since given $\overline{\mathbf{x}}{:}(A_1{\to}B) \wedge (A_2{\to}B)$ we can synthesize $\overline{\mathbf{x}} \uparrow A_1 \to B$ and $\overline{\mathbf{x}} \uparrow A_2 \to B$ using $\wedge E_1$ and $\wedge E_2$, respectively.

In the above situation, $e' = x$ is a variable, so there is a best type C—namely $\Gamma(x)$—such that if $x \uparrow C_1$ and $x \uparrow C_2$ then $x \uparrow C$, from which follows (by rules $\wedge E_{1,2}$ in the example above) $x \uparrow C_1$ and $x \uparrow C_2$. We say that x has the property of *principal synthesis*. However, it is not readily apparent which terms have this property. It holds for all variables[4]: the best type for some x is $\Gamma(x)$. On the other hand, it does not hold for many non-values: $f\ x \uparrow A_1$ and $f\ x \uparrow A_2$ do not imply $f\ x \uparrow A_1 \wedge A_2$, since the intersection introduction rule $\wedge I$ is (1) restricted to values and (2) in the checking direction. Fortunately, it does not *need* to hold for non-values: Consider $(e_1\ e_2)\ y$. Since $(e_1\ e_2)$ is not a value, y is not in evaluation position in $(e_1\ e_2)\ y$, so even in the tridirectional system, to name y we must first name $(e_1\ e_2)$. Here, the let-normal system is no more restrictive than the tridirectional system. Moreover, some values such as $()$ are checking forms and *never* synthesize, so they do not have the principal synthesis property. But again, this is fine, because we never bind values in checking form to linear variables.

All the above led us to conjecture that all *values in synthesizing form* have the principal synthesis property. The only values in synthesizing form are ordinary variables x, linear variables $\overline{\mathbf{x}}$, and annotated values $(v : As)$. For x or $\overline{\mathbf{x}}$ the principal type is simply $\Gamma(x)$ or $\Delta(\overline{\mathbf{x}})$. Unfortunately, principal types do not always exist for terms of the form $(v : As)$. For example,

$$((\lambda x.\, x) : (\vdash \mathbf{1} \to \mathbf{1}), (\vdash \mathsf{bool} \to \mathsf{bool})) \tag{5.1}$$

can synthesize $\mathbf{1} \to \mathbf{1}$, and it can synthesize $\mathsf{bool} \to \mathsf{bool}$, but it cannot synthesize their intersection, so it has no principal type.

Best typings?

We considered requiring a *best typing* among the list of typings in an annotation, where by "best" we mean "a subtype of the right hand side of all matching typings". Thus

$$((\lambda x.\, x) : (\vdash (\mathbf{1} \to \mathbf{1})), (\vdash \mathsf{bool} \to \mathsf{bool}), (\vdash (\mathbf{1} \to \mathbf{1}) \wedge (\mathsf{bool} \to \mathsf{bool})))$$

[4] In rules such as contra that do not examine their subject, this property is guaranteed by premises of the form $\Gamma \vdash e\ \mathsf{ok}$.

has a best typing because $(\mathbf{1} \to \mathbf{1}) \wedge (\mathsf{bool} \to \mathsf{bool})$ is a subtype of each of its conjuncts. One can effectively force best typings to exist by synthesizing the intersection of matching types: for (5.1) we could synthesize $(\mathbf{1} \to \mathbf{1}) \wedge (\mathsf{bool} \to \mathsf{bool})$.[5] Unfortunately, this is not enough: even if there is a best *typing*, the formulation of contextual subtyping means that there may be no best *type* arising from that typing. That is, given Γ_0, A_0 and Γ, there may be more than one A such that $(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$. In the following example, the index variable a is underdetermined—in rule $\lesssim$-ivar (Fig. 3.5), we can substitute any natural number i for a. Consequently, for every i the term synthesizes $\mathsf{list}(i) \to \mathsf{list}(i)$, but no choice of i subsumes all of these: the term cannot synthesize $\Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \to \mathsf{list}(a)$.

$$((\lambda x.\,x) : (a{:}\mathcal{N} \vdash \mathsf{list}(a) \to \mathsf{list}(a)))$$

One might call this example, like (5.1), silly: the user should simply have written

$$((\lambda x.\,x) : \ \vdash \Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \to \mathsf{list}(a))$$

Could we change the definition of contextual subtyping to ensure a unique A in $(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$? Not easily. It seems clear that contextual subtyping must depend on subtyping: $x{:}A$ supports $x{:}B$ if $A \leq B$. In practice, subtyping cannot be implemented directly from the declarative system of Figure 2.11; rather, rules such as δ generate constraints. Due to nondeterminism in the subtyping system, such as the choice between rules $\wedge L_1$ and $\wedge L_2$, the constraint generated need not be unique.

Slack bindings

Rather than try to invent some restriction on the form of typing annotations, we finesse the whole issue by "slackening" bindings of annotated values: a *slack binding* $\sim\bar{\mathbf{x}} = v$ is like an ordinary linear variable binding except that v's type need not be synthesized at its binding site, but can be synthesized at any point up to its use:

$$\frac{\Gamma; \Delta_1 \vdash v \uparrow A \qquad \Gamma; \Delta_2, \bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta_1, \Delta_2, \sim\bar{\mathbf{x}} = v \vdash e \downarrow C}\ \sim\mathsf{var} \qquad \frac{\Gamma; \Delta, \sim\bar{\mathbf{x}} = v \vdash \mathcal{Q}[\bar{\mathbf{x}}] \downarrow C}{\Gamma; \Delta \vdash \mathbf{let}\ \sim\bar{\mathbf{x}} = v\ \mathbf{in}\ \mathcal{Q}[\bar{\mathbf{x}}] \downarrow C}\ \mathsf{let}\sim$$

Wherever $\bar{\mathbf{x}}$ is in scope, we can try rule $\sim$var to synthesize a type A for v and replace $\sim\bar{\mathbf{x}} = v$ with an ordinary linear variable typing $\bar{\mathbf{x}}{:}A$.

Slack variables have the obvious shortcoming that for every slack binding we must choose when to synthesize its type, similar to how we would have to choose when to apply direct$\mathbb{L}$ to the annotated term in a naïve implementation of the tridirectional system. If all our bindings were slack we would have put ourselves in motion to no purpose, but we use slack variables for annotated values *only*. Besides, even if we came up with some wonderful solution to the issue, there would remain major sources of nondeterminism, such as the left rules which may be applied to any variables in any order. Finally, our experiments so far suggest that slack variables cause little trouble in practice; see Section 6.7.4.

[5] This would be unsound if the annotated term were not a value—it would evade the value restriction of $\wedge$I—but we are concerned only with values here.

Remark on $\top$

The above discussion of best typings centered on $\wedge$, but a similar issue arises with $\top$. Suppose $e_2 \uparrow \bot$. The let-normal translation of $((\lambda x.\, x) : \vdash \mathbf{1})\; e_2$ is

$$\textbf{let}\; \overline{\mathbf{x}} = ((\lambda x.\, x) : \vdash \mathbf{1})\; \textbf{in let}\; \overline{\mathbf{y}} = e_2\; \textbf{in let}\; \overline{\mathbf{z}} = \overline{\mathbf{x}}\; \overline{\mathbf{y}}\; \textbf{in}\; \overline{\mathbf{z}}$$

The original term is well typed in the tridirectional system: $e_2 \uparrow \bot$ is in evaluation position, so we can apply $\mathsf{direct}\mathbb{L}$ and then use $\bot\mathbb{L}$ to show that the entire term checks against any type whatsoever (this is perfectly sound: if $e_2 \uparrow \bot$ then e_2 diverges, so $v\; e_2$ diverges for all v, including $((\lambda x.\, x) : \ldots)$). But in a let-normal system without slack variables, we are stuck because $((\lambda x.\, x) : \vdash \mathbf{1})$ does not synthesize anything. A solution might be to behave as though $\top$ were always one of the annotations, so $(v : As) \uparrow \top$ for all v, but slack variables work as well:

$$\textbf{let}\; {\sim}\overline{\mathbf{x}} = ((\lambda x.\, x) : \vdash \mathbf{1})\; \textbf{in let}\; \overline{\mathbf{y}} = e_2\; \textbf{in let}\; \overline{\mathbf{z}} = \overline{\mathbf{x}}\; \overline{\mathbf{y}}\; \textbf{in}\; \overline{\mathbf{z}}$$

Here the offending term $((\lambda x.\, x) : \vdash \mathbf{1})$ is bound to $\overline{\mathbf{x}}$ but rule ${\sim}\mathsf{var}$ is never applied, since typechecking is "short-circuited" by $\bot\mathbb{L}$.

Value synthesis lemmas

Here we prove that, except for annotations $(v' : As)$, values in synthesizing form have a principal synthesis property (Lemma 5.3) and an "obligatory synthesis" property, that is, every value in synthesizing form synthesizes *something* (subject to certain conditions that hold everywhere). Both lemmas are used to prove the permutation lemmas that appear later in this chapter. But first, we need to show that $A \uparrow B$ judgments (Figure 2.8) are transitive.

The typing rules are summarized in Figures 5.3 and 5.4.

Proposition 5.2. *If* $\Gamma \vdash A \uparrow B$ *and* $\Gamma \vdash B \uparrow C$ *then* $\Gamma \vdash A \uparrow C$.

Proof. By induction on the first derivation. In case refl-$\uparrow$, $A = B$ so simply replace B with A in $\Gamma \vdash B \uparrow C$. In case $\wedge\!\uparrow_1$, we have $A = A_1 \wedge A_2$; by IH, $\Gamma \vdash A_1 \uparrow C$; by rule $\wedge\!\uparrow_1$, $\Gamma \vdash A_1 \wedge A_2 \uparrow C$, which was to be shown. The remaining cases are similar. □

Lemma 5.3 (Principal Synthesis for (Some) Values)**.** *If* $\Gamma; \Delta \vdash v \uparrow A_k$ *for* $k \in 1..n$ *where* v *does* not *have the form* $(v' : As)$ *then there exists* A *such that* $\Gamma; \Delta \vdash v \uparrow A$ *and for* $k \in 1..n$, $\Gamma \vdash A \uparrow A_k$.

Proof. By induction on the set of given typing derivations. Due to the restriction to judgments synthesizing types for values, most typing rules cannot have been used. The possible cases (for each derivation) are $\overline{\mathsf{var}}$, var, $\wedge E_{1,2}$, ΠE, and $\supset E$. ($\mathsf{ctx\text{-}anno}$ is excluded by the condition that v does not have the form $(v' : As)$.) First we treat cases in which a term-invariant rule—$\wedge E_{1,2}$, ΠE, or $\supset E$—concludes one of the derivations; without loss of generality, assume it is the first derivation.

- **Case** $\wedge E_1$: $\mathcal{D} :: \dfrac{\Gamma; \Delta \vdash v \uparrow A_1 \wedge B_1}{\Gamma; \Delta \vdash v \uparrow A_1}$

$\Gamma; \Delta \vdash v \uparrow A_1 \wedge B_1$	Subderivation
$\Gamma; \Delta \vdash v \uparrow A_k$	(for $k \in 2..n$) Derivation
$\exists A.\ \Gamma; \Delta \vdash v \uparrow A$ and $\Gamma \vdash A \uparrow A_1 \wedge B_1$ and $\Gamma \vdash A \uparrow A_k$	(for $k \in 2..n$) By IH (one derivation got smaller)
$\Gamma \vdash A_1 \wedge B_1 \uparrow A_1$	By $\wedge\!\uparrow_1$
$\Gamma \vdash A \uparrow A_1$	By Proposition 5.2

- **Cases** $\wedge E_2$, ΠE, $\supset E$: Similar to the previous case.

The remaining rules $\overline{\mathsf{var}}$ and var type distinct syntactic forms. Hence the remaining cases are:

- var *concluding all derivations.* In this case $A_k = \Gamma(x)$ for all $k \in 1..n$, and the principal type A is simply $\Gamma(x)$.

- $\overline{\mathsf{var}}$ *concluding all derivations.* Similar to the previous case. □

Lemma 5.4 (Obligatory Synthesis for (Some) Values)**.** *If v is in synthesizing form, v does not have the form $(v' : As)$, and $\Gamma \vdash v$ ok and $\Delta \Vdash v$ ok then there exists A such that $\Gamma; \Delta \vdash v \uparrow A$.*

Proof. By case analysis on v, which must have the form x or $\bar{\mathbf{x}}$.

- $\boxed{v = x}$ We have $\Gamma \vdash v\ \mathsf{ok}$, that is, $\Gamma \vdash x\ \mathsf{ok}$. Therefore $x \in \mathrm{dom}(\Gamma)$, so we can apply rule var.

- $\boxed{v = \bar{\mathbf{x}}}$ Similar to the preceding case, using $\Delta \Vdash \bar{\mathbf{x}}\ \mathsf{ok}$ and $\overline{\mathsf{var}}$. □

5.4 Introduction to the proofs

The two major results shown are *soundness*: if the let-normal form of a program is well typed in the let-normal type system, the original program is well typed in the left tridirectional system—and *completeness*: if a program is well typed in the left tridirectional type system, its translation is well typed in the let-normal type system. Once these are shown, it follows from previous results (Chapter 3) that the let-normal system is sound and complete with respect to the type assignment system in Chapter 2, for which we proved preservation and progress in a call-by-value semantics.

Soundness is relatively straightforward. This is no surprise: let-normal typechecking can be seen as left tridirectional typechecking in which the choice inherent in $\mathsf{direct}\mathbb{L}$ is forced, that is, the let-normal system is a weakening of the left tridirectional system.

On the other hand, proving completeness—that the let-normal system is not *strictly* weaker than the left tridirectional system—is quite involved; the proof overview at the start of Section 5.7 may be consulted as one reads through that section.

5.5 Preliminaries

The subtyping rules (Figure 2.11) are unchanged from previous chapters.

Throughout this chapter, we restrict the bound expression e_1 in $\mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2$ to be in synthesizing form.

We define a *let-equivalence* relation $e \equiv_{\mathsf{let}} e'$ that holds if e and e' are (possibly distinct) versions of the same direct-style term. For example:

$$\mathbf{let}\,\overline{\mathbf{x}} = x\ \mathbf{in}\ (\overline{\mathbf{x}}, ()) \equiv_{\mathsf{let}} (x, ())$$

Definition 5.5 (Let-Equivalence)**.** The relation $e \equiv_{\mathsf{let}} e'$ is defined in Figure 5.5.

Proposition 5.6. *If $v \equiv_{\mathsf{let}} e$ or $e \equiv_{\mathsf{let}} v$ then e value.*

Proof. By induction on the derivation of $v \equiv_{\mathsf{let}} e$. □

Figure 5.4 gives the rules typing the new syntactic forms. Every rule from the left tridirectional system is also a rule of the let-normal type system, with the sole exception of direct$\mathbb{L}$ which is replaced by let.

The inverse translation, the *unwinding*, is defined in Figure 5.6.

Definition 5.7. A term e is *direct style* iff it contains no **let**s and **let**∼s.

Note that by this definition, linear variables *can* appear in a direct style term! All such variables must be free since a direct style term contains no linear variable binders.

Proposition 5.8. *For all e, $\hookleftarrow(e) \equiv_{\mathsf{let}} e$ and $\hookleftarrow(e)$ is direct style.*

Proof. By induction on e.

Example case: $e = \mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2$.

$\hookleftarrow(\mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2) = [\hookleftarrow(e_1)/\overline{\mathbf{x}}]\hookleftarrow(e_2)$

$\equiv_{\mathsf{let}} \mathbf{let}\,\overline{\mathbf{x}} = \hookleftarrow(e_1)\ \mathbf{in} \hookleftarrow(e_2)$ — By defn. of $\equiv_{\mathsf{let}}$

$\hookleftarrow(e_1) \equiv_{\mathsf{let}} e_1$ — By IH

$\hookleftarrow(e_2) \equiv_{\mathsf{let}} e_2$ — By IH

☞ $\hookleftarrow(\mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2) \equiv_{\mathsf{let}} \mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2$ — Congruence of $\equiv_{\mathsf{let}}$

$\hookleftarrow(e_1), \hookleftarrow(e_2)$ are direct style — By IH

$\hookleftarrow(\mathbf{let}\,\overline{\mathbf{x}} = e_1\ \mathbf{in}\ e_2) = [\hookleftarrow(e_1)/\overline{\mathbf{x}}]\hookleftarrow(e_2)$

☞ $[\hookleftarrow(e_1)/\overline{\mathbf{x}}]\hookleftarrow(e_2)$ is direct style □

Proposition 5.9. *If e is direct style then $\hookleftarrow(e) = e$.*

Proof. By induction on e. □

Proposition 5.10. *If $e_1 \equiv_{\mathsf{let}} e_2$ then $\hookleftarrow(e_1) = \hookleftarrow(e_2)$.*

$$\frac{\Gamma(x) = A}{\Gamma; \cdot \vdash x \uparrow A}\,\textsf{var} \qquad \frac{}{\Gamma; \bar{\mathbf{x}}{:}A \vdash \bar{\mathbf{x}} \uparrow A}\,\overline{\textsf{var}}$$

$$\frac{\Gamma, x{:}A; \cdot \vdash e \downarrow B}{\Gamma; \cdot \vdash \lambda x.\, e \downarrow A \to B}\,{\to}\mathrm{I} \qquad \frac{\Gamma; \Delta_1 \vdash e_1 \uparrow A \to B \qquad \Gamma; \Delta_2 \vdash e_2 \downarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 e_2 \uparrow B}\,{\to}\mathrm{E}$$

$$\frac{\Gamma; \Delta \vdash e \uparrow A \qquad \Gamma \vdash A \leq B}{\Gamma; \Delta \vdash e \downarrow B}\,\textsf{sub} \quad \frac{\Gamma(u) = A}{\Gamma; \cdot \vdash u \uparrow A}\,\textsf{fixvar} \quad \frac{\Gamma, u{:}A; \cdot \vdash e \downarrow A}{\Gamma; \cdot \vdash \mathbf{fix}\, u.\, e \downarrow A}\,\textsf{fix} \quad \frac{}{\Gamma; \cdot \vdash () \downarrow \mathbf{1}}\,\mathbf{1}\mathrm{I}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 \downarrow A_1 \qquad \Gamma; \Delta_2 \vdash e_2 \downarrow A_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) \downarrow A_1 * A_2}\,{*}\mathrm{I} \quad \frac{\Gamma; \Delta \vdash e \uparrow A * B}{\Gamma; \Delta \vdash \mathbf{fst}(e) \uparrow A}\,{*}\mathrm{E}_1 \quad \frac{\Gamma; \Delta \vdash e \uparrow A * B}{\Gamma; \Delta \vdash \mathbf{snd}(e) \uparrow B}\,{*}\mathrm{E}_2$$

$$\frac{\overline{\Gamma} \vdash c : A \to \delta_2(i) \qquad \Gamma \vdash \delta_2(i) \leq \delta_1(j) \qquad \Gamma; \Delta \vdash e \downarrow A}{\Gamma; \Delta \vdash c(e) \downarrow \delta_1(j)}\,\delta\mathrm{I} \quad \frac{\Gamma; \Delta \vdash e \uparrow \delta(i) \qquad \Gamma; \cdot \vdash ms \downarrow_{\delta(i)} C}{\Gamma; \Delta \vdash \mathbf{case}\; e\; \mathbf{of}\; ms \downarrow C}\,\delta\mathrm{E}$$

$$\frac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash e\ \textsf{ok} \qquad \Delta \Vdash e\ \textsf{ok}}{\Gamma; \Delta \vdash e \downarrow A}\,\textsf{contra} \quad \frac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \qquad \Gamma; \Delta \vdash e \downarrow A}{\Gamma; \Delta \vdash (e : (\Gamma_0 \vdash A_0), As) \uparrow A}\,\textsf{ctx-anno}$$

$$\frac{\Gamma \vdash e\ \textsf{ok} \qquad \Delta, \bar{\mathbf{x}}{:}\bot \Vdash e\ \textsf{ok}}{\Gamma; \Delta, \bar{\mathbf{x}}{:}\bot \vdash e \downarrow C}\,\bot\mathbb{L} \quad \frac{\Gamma \vdash v\ \textsf{ok} \qquad \Delta \Vdash v\ \textsf{ok}}{\Gamma; \Delta \vdash v \downarrow \top}\,\top\mathrm{I}$$

$$\frac{\Gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}A \wedge B \vdash e \downarrow C}\,\wedge\mathbb{L}_1 \quad \frac{\Gamma; \Delta, \bar{\mathbf{x}}{:}B \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}A \wedge B \vdash e \downarrow C}\,\wedge\mathbb{L}_2$$

$$\frac{\Gamma; \Delta \vdash v \downarrow A \qquad \Gamma; \Delta \vdash v \downarrow B}{\Gamma; \Delta \vdash v \downarrow A \wedge B}\,\wedge\mathrm{I}$$

$$\frac{\Gamma; \Delta \vdash e \uparrow A \wedge B}{\Gamma; \Delta \vdash e \uparrow A}\,\wedge\mathrm{E}_1 \quad \frac{\Gamma; \Delta \vdash e \uparrow A \wedge B}{\Gamma; \Delta \vdash e \uparrow B}\,\wedge\mathrm{E}_2$$

$$\frac{\Gamma; \Delta, \bar{\mathbf{x}}{:}[i/a]A \vdash e \downarrow C \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma; \Delta, \bar{\mathbf{x}}{:}\Pi a{:}\gamma.\, A \vdash e \downarrow C}\,\Pi\mathbb{L} \quad \frac{\Gamma, a{:}\gamma; \Delta \vdash v \downarrow A}{\Gamma; \Delta \vdash v \downarrow \Pi a{:}\gamma.\, A}\,\Pi\mathrm{I} \quad \frac{\Gamma; \Delta \vdash e \uparrow \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma; \Delta \vdash e \uparrow [i/a]\, A}\,\Pi\mathrm{E}$$

$$\frac{\Gamma, a{:}\gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}\Sigma a{:}\gamma.\, A \vdash e \downarrow C}\,\Sigma\mathbb{L} \quad \frac{\Gamma; \Delta \vdash e \downarrow [i/a]\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma; \Delta \vdash e \downarrow \Sigma a{:}\gamma.\, A}\,\Sigma\mathrm{I}$$

$$\frac{\overline{\Gamma} \models P \qquad \Gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}P \supset A \vdash e \downarrow C}\,{\supset}\mathbb{L} \quad \frac{\Gamma, P; \Delta \vdash v \downarrow A}{\Gamma; \Delta \vdash v \downarrow P \supset A}\,{\supset}\mathrm{I} \quad \frac{\Gamma; \Delta \vdash e \uparrow P \supset A \qquad \overline{\Gamma} \models P}{\Gamma; \Delta \vdash e \uparrow A}\,{\supset}\mathrm{E}$$

$$\frac{\Gamma, P; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}(P \mathbin{\&} A) \vdash e \downarrow C}\,{\&}\mathbb{L} \quad \frac{\Gamma; \Delta \vdash e \downarrow A \qquad \overline{\Gamma} \models P}{\Gamma; \Delta \vdash e \downarrow P \mathbin{\&} A}\,{\&}\mathrm{I}$$

$$\frac{\Gamma; \Delta, \bar{\mathbf{x}}{:}A \vdash e \downarrow C \qquad \Gamma; \Delta, \bar{\mathbf{x}}{:}B \vdash e \downarrow C}{\Gamma; \Delta, \bar{\mathbf{x}}{:}A \vee B \vdash e \downarrow C}\,\vee\mathbb{L} \quad \frac{\Gamma; \Delta \vdash e \downarrow A}{\Gamma; \Delta \vdash e \downarrow A \vee B}\,\vee\mathrm{I}_1 \quad \frac{\Gamma; \Delta \vdash e \downarrow B}{\Gamma; \Delta \vdash e \downarrow A \vee B}\,\vee\mathrm{I}_2$$

Rules of the left tridirectional system absent in the let-normal system:

$$\left[\; \frac{\begin{array}{l} e' \text{ not a linear var} \\ \Gamma; \Delta_1 \vdash e' \uparrow^{\mathbb{L}} A \qquad \Gamma; \Delta_2, \bar{\mathbf{x}}{:}A \vdash \mathcal{E}[\bar{\mathbf{x}}] \downarrow^{\mathbb{L}} C \end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{E}[e'] \downarrow^{\mathbb{L}} C}\,\textsf{direct}\mathbb{L} \;\right]$$

Figure 5.3: Rules common to the left tridirectional and let-normal type systems

Marker version of direct$\mathbb{L}$; *replaces* direct$\mathbb{L}$

$$\frac{\Gamma; \Delta_1 \vdash e' \uparrow A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}{\Gamma; \Delta_1, \Delta_2 \vdash \textbf{let}\ \overline{\mathbf{x}} = e'\ \textbf{in}\ \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}\ \mathsf{let}$$

New rules for slack let-bindings

$$\frac{\Gamma; \Delta, \sim\overline{\mathbf{x}} = v \vdash \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}{\Gamma; \Delta \vdash \textbf{let} \sim\overline{\mathbf{x}} = v\ \textbf{in}\ \mathcal{Q}[\overline{\mathbf{x}}] \downarrow C}\ \mathsf{let}{\sim}$$

$$\frac{\Gamma; \Delta_1 \vdash v \uparrow A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta_1, \Delta_2, \sim\overline{\mathbf{x}} = v \vdash e \downarrow C}\ {\sim}\mathsf{var}$$

Figure 5.4: Typing rules new in the let-normal system

Congruence closure of:

$$\frac{}{\textbf{let}\ \overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2 \equiv_{\mathsf{let}} [e_1/\overline{\mathbf{x}}]\, e_2} \qquad \frac{}{\textbf{let} \sim\overline{\mathbf{x}} = v_1\ \textbf{in}\ e_2 \equiv_{\mathsf{let}} [v_1/\overline{\mathbf{x}}]\, e_2}$$

Figure 5.5: Relation between different versions of the same direct-style term

$$\begin{aligned}
\hookleftarrow(x) &= x & \hookleftarrow(()) &= () \\
\hookleftarrow(\overline{\mathbf{x}}) &= \overline{\mathbf{x}} & \hookleftarrow((e_1, e_2)) &= (\hookleftarrow(e_1), \hookleftarrow(e_2)) \\
\hookleftarrow(\lambda x.\, e) &= \lambda x.\, \hookleftarrow(e) & \hookleftarrow(\textbf{fst}(e)) &= \textbf{fst}(\hookleftarrow(e)) \\
\hookleftarrow(e_1\, e_2) &= \hookleftarrow(e_1) \hookleftarrow(e_2) & \hookleftarrow(\textbf{snd}(e)) &= \textbf{snd}(\hookleftarrow(e)) \\
\hookleftarrow(u) &= u & \hookleftarrow(c(e)) &= c(\hookleftarrow(e)) \\
\hookleftarrow(\textbf{fix}\ u.\, e) &= \textbf{fix}\ u.\, \hookleftarrow(e) & \hookleftarrow(\textbf{case}\ e\ \textbf{of}\ ms) &= \textbf{case} \hookleftarrow(e)\ \textbf{of} \hookleftarrow(ms) \\
\hookleftarrow(e : As) &= \hookleftarrow(e) : As & \hookleftarrow(\cdot) &= \cdot \\
& & \hookleftarrow(c(x) \Rightarrow e \mid ms) &= c(x) \Rightarrow \hookleftarrow(e) \mid \hookleftarrow(ms)
\end{aligned}$$

$$\begin{aligned}
\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2) &= [\hookleftarrow(e_1)/\overline{\mathbf{x}}] \hookleftarrow(e_2) \\
\hookleftarrow(\textbf{let} \sim\overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2) &= [\hookleftarrow(e_1)/\overline{\mathbf{x}}] \hookleftarrow(e_2)
\end{aligned}$$

Figure 5.6: Unwinding

Proof. By induction on the derivation of $e_1 \equiv_{\mathsf{let}} e_2$.

In the transitive rule (which is elided), we have subderivations of $e_1 \equiv_{\mathsf{let}} e'$ and $e' \equiv_{\mathsf{let}} e_2$. By induction on these, we get $\hookleftarrow(e_1) = \hookleftarrow(e')$ and $\hookleftarrow(e') = \hookleftarrow(e_2)$, from which $\hookleftarrow(e_1) = \hookleftarrow(e_2)$ follows by transitivity of equality. □

Proposition 5.11. *If $e_1 \equiv_{\mathsf{let}} e_2$ where e_1 and e_2 are direct style, then $e_1 = e_2$.*

Proof. By Proposition 5.10, $\hookleftarrow(e_1) = \hookleftarrow(e_2)$. Since e_1 and e_2 are direct style, by Proposition 5.9 $\hookleftarrow(e_1) = e_1$ and $\hookleftarrow(e_2) = e_2$. Thus $e_1 = e_2$. □

We defined a linearity judgment $\Delta \Vdash e\ \mathsf{ok}$ back in Chapter 3, where we had no binders for linear variables and no slack variables, and the judgment was so simple that we omitted a formal inductive definition, saying only that the judgment holds iff everything in $\mathrm{dom}(\Delta)$ appears exactly once in e and $FLV(e) \subseteq \mathrm{dom}(\Delta)$. With the addition of binders, a formal definition becomes worthwhile. However, the bulk of the definition is uninteresting, so we show only rules involving linear variables or syntactic markers, along with a couple of uninteresting rules to make the pattern clear. (Having $\bar{\mathbf{x}}{:}\top$ is a hack perpetrated to avoid defining something exactly like Δ but without types.)

See Def. 3.18.

$$\frac{\Delta_1 \Vdash e_1\ \mathsf{ok} \qquad \Delta_2, \bar{\mathbf{x}}{:}\top \Vdash e_2\ \mathsf{ok}}{\Delta_1, \Delta_2 \Vdash \mathbf{let}\ \bar{\mathbf{x}} = e_1\ \mathbf{in}\ e_2\ \mathsf{ok}} \qquad \frac{}{\bar{\mathbf{x}}{:}A \Vdash \bar{\mathbf{x}}\ \mathsf{ok}}$$

$$\frac{\Delta_1 \Vdash e_1\ \mathsf{ok} \qquad \Delta_2, \bar{\mathbf{x}}{:}\top \Vdash e_2\ \mathsf{ok}}{\Delta_1, \Delta_2 \Vdash \mathbf{let}\ {\sim}\bar{\mathbf{x}} = e_1\ \mathbf{in}\ e_2\ \mathsf{ok}}$$

$$\frac{\cdot \Vdash e\ \mathsf{ok}}{\cdot \Vdash \lambda x.\, e\ \mathsf{ok}} \qquad \frac{}{\cdot \Vdash x\ \mathsf{ok}} \qquad \frac{\Delta_1 \Vdash e_1\ \mathsf{ok} \qquad \Delta_2 \Vdash e_2\ \mathsf{ok}}{\Delta_1, \Delta_2 \Vdash (e_1, e_2)\ \mathsf{ok}}$$

Figure 5.7: Part of the new definition of the linearity judgment $\Delta \Vdash e\ \mathsf{ok}$

Proposition 5.12. *If $\Gamma; \Delta \vdash e \uparrow C$ or $\Gamma; \Delta \vdash e \downarrow C$ then $\Delta \Vdash e\ \mathsf{ok}$.*

Proof. By induction on the derivation. □

Definition 5.13. The substitution of bindings $[L]e$ is defined as follows:

$$[\cdot]e = e$$
$$[L, \bar{\mathbf{x}} = e']\, e = [L]\, [e'/\bar{\mathbf{x}}]\, e$$
$$[L, ({\sim}\bar{\mathbf{x}} = e')]\, e = [L]\, [e'/\bar{\mathbf{x}}]\, e$$

Lemma 5.14. $L\ \mathbf{in}\ e \equiv_{\mathsf{let}} [L]e$.

Proof. By induction on L. If $L = \cdot$ then $e \equiv_{\mathsf{let}} e = [\cdot]e$, which was to be shown. If $L = L', \bar{\mathbf{x}} = e'$ then:

$$\begin{aligned} L\ \mathbf{in}\ e &= L', \bar{\mathbf{x}} = e'\ \mathbf{in}\ e \\ &= L'\ \mathbf{in}\ (\bar{\mathbf{x}} = e'\ \mathbf{in}\ e) \\ &\equiv_{\mathsf{let}} L'\ \mathbf{in}\ ([e'/\bar{\mathbf{x}}]e) && \text{By defn. of } \equiv_{\mathsf{let}} \\ &\equiv_{\mathsf{let}} [L']\, ([e'/\bar{\mathbf{x}}]e) && \text{By IH} \\ &= [L]e && \text{By Def. 5.13} \end{aligned}$$

□

Let $\hookleftarrow\!\!\!\rightarrow(\mathsf{L})$ be defined in the obvious way: $\hookleftarrow\!\!\!\rightarrow(\overline{\mathbf{x}} = e), \mathsf{L}' = \overline{\mathbf{x}} = \hookleftarrow\!\!\!\rightarrow(e), \hookleftarrow\!\!\!\rightarrow(\mathsf{L}')$.

Lemma 5.15. $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e) = [\hookleftarrow\!\!\!\rightarrow(\mathsf{L})]\ \hookleftarrow\!\!\!\rightarrow(e)$.

Proof. By induction on L. ☐

Proposition 5.16. *If* $e \hookrightarrow \mathsf{L} + e'$ *then* $\mathsf{L}\ \mathbf{in}\ e'$ *is maximal, i.e.* e' *is not a* **let** *or* **let**$\sim$.

Proof. By case analysis on $e \hookrightarrow \mathsf{L} + e'$. ☐

Proposition 5.17. *Let* $\mathit{BLV}(\mathsf{L})$ *denote the linear variables bound by* L *(e.g.* $\mathit{BLV}(\overline{\mathbf{x}} = e_1, \overline{\mathbf{y}} = \overline{\mathbf{x}}\,()) = \{\overline{\mathbf{y}}\}$*). If* $\cdot \Vdash e\ \mathsf{ok}$ *and* $e \hookrightarrow \mathsf{L} + e'$ *then* $\mathit{BLV}(\mathsf{L}) = \mathit{FLV}(e')$.

Proof. By induction on $e \hookrightarrow \mathsf{L} + e'$. ☐

Proposition 5.18. *If* $e \hookrightarrow \mathsf{L} + e'$ *then* $e \equiv_{\mathsf{let}} \mathsf{L}\ \mathbf{in}\ e'$. *(Likewise, if* $ms \hookrightarrow ms'$ *then* $ms \equiv_{\mathsf{let}} ms'$*.)*

Proof. By induction on $e \hookrightarrow \mathsf{L} + e'$. We show one of the more involved cases.

- **Case**: $\mathcal{D} :: \dfrac{\check{e}_1 \hookrightarrow \mathsf{L}_1 + e_1' \qquad e_2 \hookrightarrow \mathsf{L}_2 + e_2'}{\check{e}_1 e_2 \hookrightarrow (\mathsf{L}_1, \mathsf{L}_2, \overline{\mathbf{x}} = e_1' e_2') + \overline{\mathbf{x}}}$

$\mathit{BLV}(\mathsf{L}_1), \mathit{BLV}(\mathsf{L}_2)$ disjoint	
$\mathit{FLV}(e_1') = \mathit{BLV}(\mathsf{L}_1)$	By Proposition 5.17
$\mathit{FLV}(e_1'), \mathit{BLV}(\mathsf{L}_2)$ disjoint	
$\check{e}_1 \equiv_{\mathsf{let}} \mathsf{L}_1\ \mathbf{in}\ e_1'$	By IH
$e_2 \equiv_{\mathsf{let}} \mathsf{L}_2\ \mathbf{in}\ e_2'$	By IH
$\check{e}_1 e_2 \equiv_{\mathsf{let}} (\mathsf{L}_1\ \mathbf{in}\ e_1')(\mathsf{L}_2\ \mathbf{in}\ e_2')$	By defn. of $\equiv_{\mathsf{let}}$
$\equiv_{\mathsf{let}} ([\mathsf{L}_1]e_1')(\mathsf{L}_2\ \mathbf{in}\ e_2')$	By Lemma 5.14 and defn. of $\equiv_{\mathsf{let}}$
$= [\mathsf{L}_1]\ (e_1'(\mathsf{L}_2\ \mathbf{in}\ e_2'))$	$\mathit{BLV}(\mathsf{L}_1), \mathit{FLV}(\mathsf{L}_2\ \mathbf{in}\ e_2')$ disjoint
$= [\mathsf{L}_1]\ (e_1'([\mathsf{L}_2]e_2'))$	By Lemma 5.14
$= [\mathsf{L}_1, \mathsf{L}_2]\ (e_1' e_2')$	$\mathit{BLV}(\mathsf{L}_2), \mathit{FLV}(e_1')$ disjoint
$\equiv_{\mathsf{let}} \mathsf{L}_1, \mathsf{L}_2\ \mathbf{in}\ e_1' e_2'$	By Lemma 5.14
$\equiv_{\mathsf{let}} \mathsf{L}_1, \mathsf{L}_2, \overline{\mathbf{x}} = e_1' e_2'\ \mathbf{in}\ \overline{\mathbf{x}}$	By defn. of $\equiv_{\mathsf{let}}$ ☐

Proposition 5.19. *If* $e \hookrightarrow \mathsf{L} + e'$ *then* $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e') = e$.

Proof. By Proposition 5.8, $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e') \equiv_{\mathsf{let}} \mathsf{L}\ \mathbf{in}\ e'$ and $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e')$ is direct style.

The relation $e \hookrightarrow \ldots$ is defined only when e is direct style, so e is direct style. Thus we have two direct style terms $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e')$ and e. By Proposition 5.18 $e \equiv_{\mathsf{let}} \mathsf{L}\ \mathbf{in}\ e'$. By transitivity of $\equiv_{\mathsf{let}}$, $e \equiv_{\mathsf{let}} \hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e')$. By Proposition 5.11, $\hookleftarrow\!\!\!\rightarrow(\mathsf{L}\ \mathbf{in}\ e') = e$. ☐

5.6 Soundness

In this section, we show that if a term is well typed in the let-normal system, it is well typed in the left tridirectional system.

Proposition 5.20. *If* $\Gamma \vdash e\ \mathsf{ok}$ *then* $\Gamma \vdash \hookleftarrow(e)\ \mathsf{ok}$.

Proof. By induction on e. □

Proposition 5.21. *If* $\Delta \Vdash e\ \mathsf{ok}$ *then* $\Delta \Vdash \hookleftarrow(e)\ \mathsf{ok}$.

Proof. By induction on e. □

Proposition 5.22. *If* $e\ \mathsf{value}$ *then* $\hookleftarrow(e)\ \mathsf{value}$.

Proof. By induction on e. In the case where e is a **let** $[\sim]$, use the fact that substituting a value for a value in a value yields a value. □

Definition 5.23. e is *let-respecting* if for every let-binding $\textbf{let}\ \overline{\mathbf{x}} = e_1\ \textbf{in}\ e^*$ appearing anywhere in e, there exists $\mathcal{E}$ such that $\mathcal{E}[\overline{\mathbf{x}}] = \hookleftarrow(e^*)$.

Recall that $\mathcal{C}$ denotes contexts with a hole in any position, in contrast to evaluation contexts $\mathcal{E}$ in which the hole must be in an evaluation position. For example, if $\mathcal{C} = \lambda x.\,[\,]$ then $\mathcal{C}[x] = \lambda x.\,x$, but since x is not in evaluation position in $\lambda x.\,x$, there is no $\mathcal{E}$ such that $\mathcal{E}[x] = \lambda x.\,x$.

Proposition 5.24. *For all* $\mathcal{C}$ *there exists* $\mathcal{C}'$ *such that:*
for all e *such that no linear variable free in* e *is bound in* $\mathcal{C}$,
$\hookleftarrow(\mathcal{C}[e]) = \mathcal{C}'[\hookleftarrow(e)]$.
Moreover, if $\mathcal{C}$ *is an evaluation context (i.e.* $\mathcal{E} = \mathcal{C}$*) and* $\mathcal{C}[e]$ *is let-respecting, then* $\mathcal{C}'$ *is also an evaluation context, that is, there exists* $\mathcal{E}' = \mathcal{C}'$ *such that* $\hookleftarrow(\mathcal{E}[e]) = \mathcal{E}'[\hookleftarrow(e)]$.

Remark 5.25. The condition on e forbids situations such as $\mathcal{C} = \textbf{let}\ \overline{\mathbf{x}} = x\ \textbf{in}\ [\,]$ with $e = \overline{\mathbf{x}}$, where $\overline{\mathbf{x}}$ would be captured and the proposition would not hold: $\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = x\ \textbf{in}\ \overline{\mathbf{x}}) = x$, but there is no $\mathcal{C}'$ such that $x = \mathcal{C}'[\hookleftarrow(\overline{\mathbf{x}})] = \mathcal{C}'[\overline{\mathbf{x}}]$.

Proof. By induction on $\mathcal{C}$. Most cases are straightforward. In the case where $\mathcal{C} = \textbf{let}\ \overline{\mathbf{x}} = e_1\ \textbf{in}\ \mathcal{C}_2$, let $\mathcal{C}' = [\hookleftarrow(e_1)/\overline{\mathbf{x}}]\ \mathcal{C}_2$; note that $\overline{\mathbf{x}}$ is bound in $\mathcal{C}[e]$ so it cannot appear free in e.

In the 'Moreover' part, we show the two interesting cases:

- $\boxed{\mathcal{E} = \textbf{let}\ \overline{\mathbf{x}} = \mathcal{E}_0\ \textbf{in}\ e_2}$ $\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = \mathcal{E}_0[e]\ \textbf{in}\ e_2) = [\hookleftarrow(\mathcal{E}_0[e])/\overline{\mathbf{x}}]\ (\hookleftarrow(e_2))$. It is given that $\mathcal{E}[e]$ is let-respecting, so $\hookleftarrow(e_2) = \mathcal{E}_2[\overline{\mathbf{x}}]$ for some $\mathcal{E}_2$. By IH, $\hookleftarrow(\mathcal{E}_0[e]) = \mathcal{E}_0'[\hookleftarrow(e)]$ for some $\mathcal{E}_0'$. Assuming $\overline{\mathbf{x}}$ appears linearly, $\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = \mathcal{E}_0[e]\ \textbf{in}\ e_2) = [\mathcal{E}_0'[\hookleftarrow(e)]/\overline{\mathbf{x}}]\ \mathcal{E}_2[\overline{\mathbf{x}}] = \mathcal{E}_2[\mathcal{E}_0'[\hookleftarrow(e)]]$. Let $\mathcal{E}' = \mathcal{E}_2[\mathcal{E}_0']$.

- $\boxed{\mathcal{E} = \textbf{let}\ \overline{\mathbf{x}} = v\ \textbf{in}\ \mathcal{E}_0}$ $\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = v\ \textbf{in}\ \mathcal{E}_0[e]) = [\hookleftarrow(v)/\overline{\mathbf{x}}]\ (\hookleftarrow(\mathcal{E}_0[e]))$.

  By IH, this is equal to $[\hookleftarrow(v)/\overline{\mathbf{x}}]\ \mathcal{E}_0'[\hookleftarrow(e)]$ for some $\mathcal{E}_0'$. By Proposition 5.22, $\hookleftarrow(v)$ is a value. Replacing the value ($\overline{\mathbf{x}}$) with the value ($\hookleftarrow(v)$) cannot affect whether a subterm ($\hookleftarrow(e)$) is in evaluation position. Therefore $\hookleftarrow(\textbf{let}\ \overline{\mathbf{x}} = v\ \textbf{in}\ \mathcal{E}_0[e]) = \mathcal{E}'[\hookleftarrow(e)]$, where $\mathcal{E}' = [\hookleftarrow(v)/\overline{\mathbf{x}}]\ \mathcal{E}_0'$. □

Proposition 5.26. *For all $\mathcal{C}$, if $\hookleftarrow\!\!\rightarrow(e_1) = \hookleftarrow\!\!\rightarrow(e_2)$ then $\hookleftarrow\!\!\rightarrow(\mathcal{C}[e_1]) = \hookleftarrow\!\!\rightarrow(\mathcal{C}[e_2])$.*

Proof. By induction on $\mathcal{C}$. □

Lemma 5.27. *If $\check{e} \hookrightarrow L + e'$ then e' value.*

Proof. By induction on the derivation. We consider only rules deriving a conclusion where the domain of the translation is a pre-value. In most of those cases, $e' = \overline{\mathbf{x}}$ so e' value is immediate. Two cases require a little work:

- In the rule deriving $(\check{e}_1, e_2) \hookrightarrow (L_1, L_2) + (e_1', e_2')$ we have $\check{e} = (\check{e}_1, e_2)$; since $(\check{e}_1, e_2)$ prevalue, according to the grammar of pre-values it must be the case that e_2 prevalue; by IH, e_2' value. Also by IH, e_1' value. Therefore $e' = (e_1', e_2')$ value.

- In the rule deriving $c(\check{e}) \hookrightarrow L + c(e')$, we must apply the IH. □

Lemma 5.28. *If $e \hookrightarrow L + e'$ then L **in** e' is let-respecting.*

Proof. By induction on the derivation. If e has any of the forms $\{x, \lambda x.\, e_0, u, \textbf{fix}\ u.\, e_0, (e_0 : As), (v_0 : As), \textbf{fst}(e_0), \textbf{snd}(e_0), (), \overline{\mathbf{x}}, c(e_0), \textbf{case}\ e_0\ \textbf{of}\ ms\}$ the case for the corresponding rule is fairly straightforward. The cases for $e = \widehat{e}_1 e_2$ and $e = (\widehat{e}_1, e_2)$ are similar. However, if $e = \check{e}_1 e_2$ or $e = (\check{e}_1, e_2)$ we must take care: the corresponding rule interpolates the bindings from the translation of e_2 between L_1 and e_1' (where $\check{e}_1 \hookrightarrow L_1 + e_1'$). We show the $e = (\check{e}_1, e_2)$ case in full:

- **Case**: $\mathcal{D} :: \dfrac{\check{e}_1 \hookrightarrow L_1 + e_1' \qquad e_2 \hookrightarrow L_2 + e_2'}{\check{e}_1 e_2 \hookrightarrow L_1, L_2, \overline{\mathbf{x}} = e_1' e_2' + \overline{\mathbf{x}}}$

$\check{e}_1 \hookrightarrow L_1 + e_1'$	Subd.
$(L_1\ \textbf{in}\ e_1')$ is let-respecting	By IH
$e_2 \hookrightarrow L_2 + e_2'$	Subd.
$(L_2\ \textbf{in}\ e_2')$ is let-respecting	By IH

First we show that the bindings in L_1 are let-respecting.

  – Consider each $\overline{\mathbf{y}} = e_y$ in L_1, where $L_1 = L_{11}, \overline{\mathbf{y}} = e_y, L_{12}$. By IH, L_1 **in** e_1' is let-respecting; in particular, $\hookleftarrow\!\!\rightarrow(L_{12}\ \textbf{in}\ e_1') = \mathcal{E}[\overline{\mathbf{y}}]$ for some $\mathcal{E}$. By Lemma 5.15, we have $[\hookleftarrow\!\!\rightarrow(L_{12})] \hookleftarrow\!\!\rightarrow(e_1') = \mathcal{E}[\overline{\mathbf{y}}]$. This will be useful shortly.

  Next, by the same lemma, we have

$$\begin{aligned}\hookleftarrow\!\!\rightarrow(L_{12}, L_2, \overline{\mathbf{x}} = e_1' e_2'\ \textbf{in}\ \overline{\mathbf{x}}) &= [\hookleftarrow\!\!\rightarrow(L_{12})]\ [\hookleftarrow\!\!\rightarrow(L_2)]\ [\hookleftarrow\!\!\rightarrow(e_1' e_2')/\overline{\mathbf{x}}]\ \overline{\mathbf{x}} \\ &= [\hookleftarrow\!\!\rightarrow(L_{12})]\ [\hookleftarrow\!\!\rightarrow(L_2)]\ \hookleftarrow\!\!\rightarrow(e_1' e_2')\end{aligned}$$

  Since variables in $BLV(L_k)$ are free only in e_k', and in particular $BLV(L_{12}) \cap e_2' = \{\}$ and $BLV(L_2) \cap e_1' = \{\}$, we get

$$\big([\hookleftarrow\!\!\rightarrow(L_{12})] \hookleftarrow\!\!\rightarrow(e_1')\big)\ \big([\hookleftarrow\!\!\rightarrow(L_2)] \hookleftarrow\!\!\rightarrow(e_2')\big)$$

We obtained $[\hookleftarrow(L_{12})]\hookleftarrow(e_1') = \mathcal{E}[\overline{\mathbf{y}}]$ earlier, so we get

$$\hookleftarrow(L_{12}, L_2, \overline{\mathbf{x}} = e_1' e_2' \textbf{ in } \overline{\mathbf{x}}) = \mathcal{E}[\overline{\mathbf{y}}]\left([\hookleftarrow(L_2)]\hookleftarrow(e_2')\right)$$

Let $\mathcal{E}' = \mathcal{E}[\overline{\mathbf{y}}]\,([\hookleftarrow(L_2)]\hookleftarrow(e_2'))$. This is the evaluation context that shows the binding of $\overline{\mathbf{y}}$ is let-respecting.

Showing that the bindings in L_2 are let-respecting starts out similarly, but then hinges on $\check{e}_1$ being a pre-value, which allows us to show e_1' value:

– Consider each $\overline{\mathbf{y}} = e_y$ in L_2, where $L_2 = L_{21}, \overline{\mathbf{y}} = e_y, L_{22}$. By IH, L_2 **in** e_2' is let-respecting; in particular, $\hookleftarrow(L_{22} \textbf{ in } e_2') = \mathcal{E}[\overline{\mathbf{y}}]$ for some $\mathcal{E}$. By Lemma 5.15, we have $[\hookleftarrow(L_{22})]\hookleftarrow(e_2') = \mathcal{E}[\overline{\mathbf{y}}]$. This will be useful shortly.
  Next, by the same lemma, we have $\hookleftarrow(L_{22}, \overline{\mathbf{x}} = e_1' e_2' \textbf{ in } \overline{\mathbf{x}}) = [\hookleftarrow(L_{22})]\,[\hookleftarrow(e_1' e_2')/\overline{\mathbf{x}}]\,\overline{\mathbf{x}} = [\hookleftarrow(L_{22})]\hookleftarrow(e_1' e_2')$. Since variables in $BLV(L_k)$ are free only in e_k', and in particular $BLV(L_{22}) \cap e_2' = \{\}$, we get
  $$\left(\hookleftarrow(e_1')\right)\left([\hookleftarrow(L_{22})]\hookleftarrow(e_2')\right)$$
  We have $\check{e}_1 \hookrightarrow L_1 + e_1'$, so e_1' value by Lemma 5.27. By Proposition 5.22, $\hookleftarrow(e_1')$ value. Earlier, we obtained $[\hookleftarrow(L_{22})]\hookleftarrow(e_2') = \mathcal{E}[\overline{\mathbf{y}}]$, so let $\mathcal{E}' = \hookleftarrow(e_1')\,\mathcal{E}[\overline{\mathbf{y}}]$; this is the evaluation context that shows the binding of $\overline{\mathbf{y}}$ is let-respecting. □

Lemma 5.29. *If $\mathcal{C}[e]$ is let-respecting and $FLV(e) \subseteq FLV(\mathcal{C}[e])$, then $\mathcal{C}[\overline{\mathbf{x}}]$ is let-respecting.*

Proof. By induction on $\mathcal{C}$. Most cases are straightforward; if $\mathcal{C} = \textbf{let}\,\overline{\mathbf{y}} = e_1 \textbf{ in}\, \mathcal{C}'$ we need to show that $\overline{\mathbf{y}}$ is in evaluation position in $\hookleftarrow(\mathcal{C}'[e])$. By Proposition 5.24 there exists $\mathcal{C}''$ such that $\hookleftarrow(\mathcal{C}'[e]) = \mathcal{C}''[\hookleftarrow(e)]$ and $\hookleftarrow(\mathcal{C}'[\overline{\mathbf{x}}]) = \mathcal{C}''[\overline{\mathbf{x}}]$. It is given that $\mathcal{C}[e] = \textbf{let}\,\overline{\mathbf{y}} = e_1 \textbf{ in}\,\mathcal{C}'[e]$ is let-respecting so $\mathcal{C}''[\hookleftarrow(e)] = \mathcal{E}[\overline{\mathbf{y}}]$ for some $\mathcal{E}$. Since $FLV(e) \subseteq FLV(\mathcal{C}[e])$ and $\overline{\mathbf{y}} \notin FLV(\mathcal{C}[e])$ (it is bound inside $\mathcal{C}[e]$), we have $\overline{\mathbf{y}} \notin FLV(e)$. Then $\overline{\mathbf{y}} \notin FLV(\hookleftarrow(e))$. Replacing a subterm with a value ($\overline{\mathbf{x}}$) in $\mathcal{E}[\overline{\mathbf{y}}]$ must yield a term of the form $\mathcal{E}'[\overline{\mathbf{y}}]$. By IH, $\mathcal{C}'[\overline{\mathbf{x}}]$ is let-respecting. □

Soundness is now quite easy to formulate and prove.

Theorem 5.30 (Let-Normal Soundness). *If $\Gamma; \Delta \vdash e \underline{\downarrow^{\mathsf{let}}\uparrow^{\mathsf{let}}}\, C$ where e is let-respecting then $\Gamma; \Delta \vdash \hookleftarrow(e) \underline{\downarrow^{\mathbb{L}}\uparrow^{\mathbb{L}}}\, C$.*

Proof. By induction on the given derivation. Since the two systems share most of their rules, most of the cases are straightforward. For the remaining cases, we proceed as follows:

- **Case** let: Apply the IH to each premise. Since e is let-respecting, there exists an evaluation context $\mathcal{E}$ such that $\hookleftarrow(e) = \mathcal{E}[e']$. Apply direct$\mathbb{L}$.
- **Cases** contra, $\bot\mathbb{L}$: Use Propositions 5.20 and 5.21.
- **Case** $\top$I: Use Propositions 5.20, 5.21, and 5.22. □

Corollary 5.31. *(Let-Normal Soundness) If $e \hookrightarrow L + e'$ and $\cdot; \cdot \vdash L \textbf{ in } e' \downarrow^{\mathsf{let}} C$ then $\cdot; \cdot \vdash e \downarrow^{\mathbb{L}} C$.*

Proof. By Lemma 5.28, L **in** e' is let-respecting. By Theorem 5.30, $\cdot; \cdot \vdash \hookleftarrow(L \textbf{ in } e') \downarrow^{\mathbb{L}} C$. Substituting e for $\hookleftarrow(L \textbf{ in } e')$, justified by Proposition 5.19, yields the result. □

5.7 Completeness

In this section, we show that given a term e that is well typed in the left tridirectional type system, the let-normal translation L **in** e' where $e \hookrightarrow \mathsf{L} + e'$ is well typed in the let-normal type system. To be precise, given a derivation $\mathcal{D}$ deriving $\Gamma; \Delta \vdash e \downarrow^{\mathbb{L}} C$ we must construct a derivation $\Gamma; \Delta \vdash \mathsf{L}$ **in** $e' \downarrow^{\mathsf{let}} C$, where $e \hookrightarrow \mathsf{L} + e'$. Attempts to prove this in a straightforward way, that is, by induction on the derivation, failed: rule $\mathsf{direct}\mathbb{L}$ means that the relationship between the "shapes" of $\mathcal{D}$ and e is nontrivial. Nor is $\mathsf{L} + e'$ trivially compositional in e: in particular, for a given subterm of e there is not always a subterm of $\mathsf{L} + e'$ corresponding to it.

Instead, the completeness proof proceeds as follows:

1. *Mark* e with **let** bindings according to the given derivation: wherever $\mathsf{direct}\mathbb{L}$ is used add a corresponding **let**. However, if $\wedge$I or another subject-duplicating rule is used, the subderivations need not apply $\mathsf{direct}\mathbb{L}$ in the same way, resulting in distinct terms to which $\wedge$I cannot be applied. In those cases we use the second step *Transform* inductively to obtain typing derivations for the canonical version of the subterm, to which $\wedge$I can be applied.

   The main component of this step is Lemma 5.79, which produces a let-system typing derivation for a new term. This term is not necessarily in canonical let-normal form. For example, if the original left tridirectional typing derivation for $\lambda x.\, x$ does not use $\mathsf{direct}\mathbb{L}$ at all, no **let** bindings are created, which does not match the canonical let-normal translation $\lambda x.\,$ **let** $\overline{\mathbf{x}} = x$ **in** $\overline{\mathbf{x}}$.

2. *Transform* the marked term into $\mathsf{L} + e'$ in small steps, adding or moving one **let** at a time, systematically approaching $\mathsf{L} + e'$. The main component of this step is Lemma 5.78, which relies on the lemmas in Section 5.7.4 to show that types are preserved as **let**s are added and moved. We define a syntactic measure (a tuple of natural numbers) that quantifies how different a term is from $\mathsf{L} + e'$; each **let**-manipulating step reduces the measure, bringing the term closer to $\mathsf{L} + e'$. The last piece of this puzzle is to check that when the measure is all zeroes, the term *is* $\mathsf{L} + e'$, shown by Lemma 5.72.

The completeness theorem itself (5.80) simply applies Lemmas 5.79 and 5.78 (and a final bit of tedious syntactic reasoning).

The remainder of this chapter consists of proofs and the infrastructure necessary for them. It may be helpful at this point to skim Lemma 5.79 and Theorem 5.80.

5.7.1 Let-free paths and the 'precedes' relation

A *let-free viable path* is one that (a) skips over no **let** and (b) leads to a hole $[]$ that, if replaced by a term in synthesizing form, can be named by $\mathsf{direct}\mathbb{L}$, perhaps only after applying $\mathsf{direct}\mathbb{L}$ to other subterms. For example, $(\lambda x.\, e)\ []$ is a let-free viable path ($\mathsf{direct}\mathbb{L}$ could be applied immediately with $\mathcal{E} = (\lambda x.\, e)\ []$), as is $((e_1 e_2)[]$ ($\mathsf{direct}\mathbb{L}$ can be applied first with $\mathcal{E}_1 = ([])\cdots$, yielding $\overline{\mathbf{x}}\ []$; $\mathsf{direct}\mathbb{L}$ can then be applied with $\mathcal{E}_2 = \overline{\mathbf{x}}\ []$).

Definition 5.32 (Let-Free Viable Paths)**.**

$$\begin{aligned}\mathcal{W} \ ::= \ & [] \mid \mathcal{W}\, e \mid \check{e}\, \mathcal{W} \mid (\mathcal{W}, e) \mid (\check{e}, \mathcal{W}) \mid \mathbf{fst}(\mathcal{W}) \mid \mathbf{snd}(\mathcal{W}) \\ & \mid (\mathcal{W} : \mathsf{As}) \mid c(\mathcal{W}) \mid \mathbf{case}\ \mathcal{W}\ \mathbf{of}\ ms \mid \mathbf{let}\ \overline{\mathbf{x}} = \mathcal{W}\ \mathbf{in}\ e\end{aligned}$$

Lemma 5.33. *If $\mathcal{E} = \mathcal{W}$ and $\mathcal{W}[\textbf{let}\,\overline{\mathbf{x}} = e_2\ \textbf{in}\ e_3]$ is let-respecting then $\mathcal{W}[e_3]$ is let-respecting.*

Proof. By induction on $\mathcal{W}$. The only vaguely interesting case is when $\mathcal{W} = \textbf{let}\,\overline{\mathbf{y}} = \mathcal{W}'\ \textbf{in}\ e_4$. By induction, $\mathcal{W}'[e_3]$ is let-respecting. Since the given term is let-respecting, there exists $\mathcal{E}_4$ such that $e_4 = \mathcal{E}_4[\overline{\mathbf{y}}]$; also, e_4 is let-respecting. Therefore $\textbf{let}\,\overline{\mathbf{y}} = \mathcal{W}'[e_3]\ \textbf{in}\ e_4$ is let-respecting. □

Definition 5.34. A subterm e' of e is in *let-free viable position* if there exists $\mathcal{W}$ such that $e = \mathcal{W}[e']$.
A variable $\overline{\mathbf{x}}$ is in *let-free viable position* [in its scope] if, in $\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2$, there exists $\mathcal{W}$ such that $e_2 = \mathcal{W}[\overline{\mathbf{x}}]$.

Note that the languages defined by the various contexts satisfy $\begin{matrix}\mathfrak{L}(\mathcal{E})\\ \mathfrak{L}(\mathcal{W})\end{matrix} \subset \mathfrak{L}(\mathcal{Q}) \subset \mathfrak{L}(\mathcal{C})$ where all inclusions are proper. (It is not the case that $\mathfrak{L}(\mathcal{E}) \subseteq \mathfrak{L}(\mathcal{W})$, since $\textbf{let}\,\overline{\mathbf{x}} = v\ \textbf{in}\ []$ is an $\mathcal{E}$ but not a $\mathcal{W}$.)
We also say what it means for a term to precede another:

Definition 5.35. Given e_1 and e_2 such that $\mathcal{C}_1[e_1] = \mathcal{C}_2[e_2] = e$, we say that e_1 *precedes* e_2 in e if e_1 appears to the left of e_2 when the term is written out in the usual way.

5.7.2 Properties of $\hookrightarrow$

Here we show that $\hookrightarrow$ is a function, so there is a single canonical let-normal version of any direct-style term, and that $\hookrightarrow$ is injective, so no distinct let-normal terms share "common ancestry".

Proposition 5.36. *The $\hookrightarrow$ translation is a function: If $e \hookrightarrow L_1 + e_1$ and $e \hookrightarrow L_2 + e_2$ then $L_1 = L_2$ and $e_1 = e_2$.*

Proof. By induction on the derivation of the judgments. □

The following proposition is used only in the proof of Proposition 5.38.

Proposition 5.37. *If $e \hookrightarrow L + e'$ then e' is a linear variable if and only if* e prevalue.

Proof. By inspection of each rule's conclusion. □

Proposition 5.38. *The $\hookrightarrow$ translation is injective: If $e_1 \hookrightarrow L + e'$ and $e_2 \hookrightarrow L + e'$ then $e_1 = e_2$.*

Proof. By induction on the derivation of the judgments.
Except for those rules with a pair or application on the left hand side of the conclusion, all the right hand sides are syntactically disjoint: for instance, the only rule concluding a judgment with $(\ldots, \overline{\mathbf{x}} = (\cdots : As)) + \overline{\mathbf{x}}$ on the right is the rule with $(e : As)$ on the left. So (again, leaving out the pair and application rules) whatever rule derived $e_1 \hookrightarrow L + e'$ also derived $e_2 \hookrightarrow L + e'$.
The two rules for application are obviously syntactically disjoint in the r.h.s. of their conclusion from all the others, but not from each other. However, again the same rule must conclude both derivations. Suppose, for a contradiction, that the anti-value rule concludes $\widehat{e}_{11} e_{12} \hookrightarrow L + e'$ and the pre-value rule concludes $\check{e}_{21} e_{22} \hookrightarrow L + e'$. In both rules, the last binding in L has the form $\overline{\mathbf{x}} = e_1' \ldots$; the $\ldots$ might vary but the e_1' in the first rule is identical to the e_1' in the second.[6] In the

[6] Note that to apply the IH here, we would need to prove that the '$\ldots$' are actually the same.

first rule, we have a premise $\widehat{e}_{11} \hookrightarrow \cdots + e_1'$. By Proposition 5.37, e_1' is not a linear variable. In the second rule, we have a premise $\check{e}_{21} \hookrightarrow \cdots + e_2'$. But by Proposition 5.37, e_1' is a linear variable, a contradiction.

The reasoning for the pair rules is similar. □

5.7.3 Position and ordering of let-bindings

We give a number of tedious syntactic definitions. The intuition about *roots* (Definition 5.41) is that a root is somewhere that the canonical translation $\hookrightarrow$ may place a sequence of let-bindings. In the course of letification (Corollary 5.49), we create **let**s that are not at roots—these are called *prickly*—and permute them outward (Corollary 5.63) until they are.

For similar reasons, we call terms like $\textbf{let}\,\overline{\mathbf{x}} = (v_1 : As)\ \textbf{in}\ e_2$ *brittle*, since they need to be slackened (changed to $\textbf{let}\,{\sim}\overline{\mathbf{x}} = (v_1 : As)\ \textbf{in}\ e_2$, using Corollary 5.51) to obtain a program in canonical let-normal form.

Finally, even when all the prickly bindings are at roots and all the brittle bindings have been slackened, individual bindings in a sequence may appear *transposed*, as in $\textbf{let}\,\overline{\mathbf{y}} = y\ \textbf{in}\ \textbf{let}\,\overline{\mathbf{x}} = x\ \textbf{in}\ (\overline{\mathbf{x}}, \overline{\mathbf{y}})$. The unwinding of that term is (x, y), but the canonical translation is $\textbf{let}\,\overline{\mathbf{x}} = x\ \textbf{in}\ \textbf{let}\,\overline{\mathbf{y}} = y\ \textbf{in}\ (\overline{\mathbf{x}}, \overline{\mathbf{y}})$, with linear variables used in the same order they are bound. So the bindings must be swapped (with Lemma 5.69).

Definition 5.39 (Colocation)**.** A set of let-bindings is *colocated* if there is a subterm $L\ \textbf{in}\ e'$ such that every binding in the set is in L.

Definition 5.40. A decomposition $L\ \textbf{in}\ e$ is *maximal* iff e is not any kind of **let**.

Definition 5.41. A term e' such that $\mathcal{C}[e'] = e$ is a *root in* e if $e' = e$ or e' is a case arm or the body of a λ or **fix** (i.e. if $\mathcal{C} = [\,]$ or $\mathcal{C} = \mathcal{C}'[\textbf{case}\ e\ \textbf{of}\ \dots \mid \dots \Rightarrow [\,] \mid \dots]$ or $\mathcal{C} = \lambda x.\,[\,]$ or $\mathcal{C} = \textbf{fix}\ u.\,[\,]$).

We call a binding that is not part of a leftmost sequence of bindings at some root *prickly*:

Definition 5.42. Given a term $e = \mathcal{C}[e_0]$ where $e_0 = \textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2$, the binding $\overline{\mathbf{x}} = e_1$ is *prickly* iff there exists $\mathcal{W} \neq [\,]$ and $\mathcal{C}'$ such that $\mathcal{C} = \mathcal{C}'[\mathcal{W}]$.

Remark 5.43. For example, the let-binding in $x\ (\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2)$ is prickly (with $\mathcal{W} = x[\,]$). On the other hand, the binding in $(\textbf{case}\ e_0\ \textbf{of}\ ms)\ (\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ e_2)$ is not, since $\mathcal{W}$ paths cannot cross anti-values such as $\textbf{case}\ e_0\ \textbf{of}\ ms$.

Also note that the canonical translation $\hookrightarrow$ does not create prickly bindings.

Definition 5.44. A subterm e' of an expression e is *brittle* if $e' = \textbf{let}\,\overline{\mathbf{x}} = (v_1 : As)\ \textbf{in}\ e_2$ and, in $\hookleftarrow\!(e)$, the unwinding of v_1 is a value.

Remark 5.45. Note that the unwinding of the value v_1 is not necessarily a value, as in

$$e \;=\; \textbf{let}\,\overline{\mathbf{y}} = y\,z\ \textbf{in}\ \overbrace{\textbf{let}\,\overline{\mathbf{x}} = (\underbrace{(\overline{\mathbf{y}}, ())}_{v_1} : As)\ \textbf{in}\ e_2}^{e'}$$

$$\hookleftarrow\!(e) \;=\; ((\underbrace{y\,z}_{\text{unwinding of }\overline{\mathbf{x}}}, ()) : As)$$

Definition 5.46 (Transposition)**.** A pair of linear variables $\langle \overline{\mathbf{x}}, \overline{\mathbf{y}} \rangle$ is *transposed* iff (1) the bindings of $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ are colocated, (2) in the unwinding, $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{y}}$, (3) $\overline{\mathbf{x}}$ is bound after $\overline{\mathbf{y}}$.

5.7.4 Type preservation lemmas

In this section, we prove that each of the following operations, when applied to a well typed term, yields a well typed term:

- adding a binding for a synthesizing subterm (*letification*)
- changing a **let** binding into a **let**~ binding (*slackening*)
- moving a **let** or **let**~, subject to certain conditions (*permutation* and *value permutation*)

The subsections are in the order in which these lemmas are used in the proof of Lemma 5.78: we first bind all synthesizing subterms without bindings, then slacken some of the bindings, then reorder the bindings to ensure that subterms are bound in the order they are used.

In this section, we silently use the fact that terms never contain bindings of the form $\textbf{let}\,\overline{\mathbf{x}} = \overline{\mathbf{y}}\,\textbf{in}\,e$. The reader can verify that no derivations involving such terms ever arise.

Letification

The letification lemma shows that adding a **let** binding a term in synthesizing form preserves typing.

Given a term $\mathcal{E}[e']$ where e' is in synthesizing form, the binding is inserted at the *nearest checking position*. That is, if e' is in a checking position the binding is inserted at that point: the term becomes $\mathcal{E}[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\overline{\mathbf{x}}]$. Otherwise, we go up the syntax tree toward the root until a checking position is reached. For example, in $(x(y), z)$ the subterm $e' = x$ is in a synthesizing position in $x(y)$, so we insert the binding at the nearest checking position, yielding $(\textbf{let}\,\overline{\mathbf{x}} = x\,\textbf{in}\,\overline{\mathbf{x}}(y), z)$.

In the example $(x(y), z)$, we could also insert the binding outside the pair: $\textbf{let}\,\overline{\mathbf{x}} = x\,\textbf{in}\,(\overline{\mathbf{x}}(y), z)$. However, the proof requires that there be no choice in the matter: in $\wedge$I, $\vee\mathbb{L}$ and $\wedge$-ct the term is duplicated in both premises, so in those cases we need to apply the induction hypothesis twice. If we do not fix the position of the new binding, applying the IH twice could result in different terms, leaving us unable to reapply the rule.

The nearest checking position function $\langle\downarrow\rangle(-)$ is defined in Figure 5.8.

Lemma 5.47 (Letification)**.** *Assume e' is in synthesizing form, but is not a linear variable.*

(1) If $\Gamma; \Delta \vdash \mathcal{E}[e'] \downarrow C$ then $\Gamma; \Delta \vdash \mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]] \downarrow C$ where $\mathcal{E}' = \langle\downarrow\rangle(\mathcal{E})$ and $\mathcal{E} = \mathcal{E}'[\mathcal{E}'']$.

(2) If $\Gamma; \Delta \vdash \mathcal{E}[e'] \uparrow C$ then either

(a) $\mathcal{E} = []$ (and therefore $\langle\downarrow\rangle(\mathcal{E}) = []$) and there exist A', Δ_1, Δ_2 such that $\Delta = (\Delta_1, \Delta_2)$ and $\Gamma; \Delta_1 \vdash e' \uparrow A'$ and $\Gamma; \Delta_2, \overline{\mathbf{x}}{:}A' \vdash \mathcal{E}[\overline{\mathbf{x}}] \uparrow C$, or

(b) $\mathcal{E} \neq []$ and $\Gamma; \Delta \vdash \mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]] \uparrow C$ where $\mathcal{E}' = \langle\downarrow\rangle(\mathcal{E})$ and $\mathcal{E} = \mathcal{E}'[\mathcal{E}'']$.

$$
\begin{aligned}
\langle\downarrow\rangle([\,]) &= [\,] \\
\langle\downarrow\rangle((\mathcal{E}, e)) &= (\langle\downarrow\rangle(\mathcal{E}), e) \\
\langle\downarrow\rangle((v, \mathcal{E})) &= (v, \langle\downarrow\rangle(\mathcal{E})) \\
\langle\downarrow\rangle(\mathcal{E} e) &= \langle\downarrow\rangle(\mathcal{E})\, e && \text{if } \langle\downarrow\rangle(\mathcal{E}) \neq [\,], \text{ otherwise } [\,] \\
\langle\downarrow\rangle(v\mathcal{E}) &= v\, \langle\downarrow\rangle(\mathcal{E}) \\
\langle\downarrow\rangle(\textbf{fst}(\mathcal{E})) &= \textbf{fst}(\langle\downarrow\rangle(\mathcal{E})) && \text{if } \langle\downarrow\rangle(\mathcal{E}) \neq [\,], \text{ otherwise } [\,] \\
\langle\downarrow\rangle(\textbf{snd}(\mathcal{E})) &= \textbf{snd}(\langle\downarrow\rangle(\mathcal{E})) && \text{if } \langle\downarrow\rangle(\mathcal{E}) \neq [\,], \text{ otherwise } [\,] \\
\langle\downarrow\rangle(\mathcal{E} : As) &= (\langle\downarrow\rangle(\mathcal{E}) : As) \\
\langle\downarrow\rangle(c(\mathcal{E})) &= c(\langle\downarrow\rangle(\mathcal{E})) \\
\langle\downarrow\rangle(\textbf{case}\ \mathcal{E}\ \textbf{of}\ ms) &= \textbf{case}\ \langle\downarrow\rangle(\mathcal{E})\ \textbf{of}\ ms && \text{if } \langle\downarrow\rangle(\mathcal{E}) \neq [\,], \text{ otherwise } [\,] \\
\langle\downarrow\rangle(\textbf{let}\ \overline{\mathbf{x}} = \mathcal{E}\ \textbf{in}\ e) &= \textbf{let}\ \overline{\mathbf{x}} = \langle\downarrow\rangle(\mathcal{E})\ \textbf{in}\ e && \text{if } \langle\downarrow\rangle(\mathcal{E}) \neq [\,], \text{ otherwise } [\,] \\
\langle\downarrow\rangle(\textbf{let}\ \overline{\mathbf{x}} = v\ \textbf{in}\ \mathcal{E}) &= \textbf{let}\ \overline{\mathbf{x}} = v\ \textbf{in}\ \langle\downarrow\rangle(\mathcal{E})
\end{aligned}
$$

Figure 5.8: The nearest checking position function $\langle\downarrow\rangle(-)$

Remark 5.48. In part (2), we have a synthesizing term $\mathcal{E}[e']$: following some path from the root of $\mathcal{E}[e']$, we reach a subterm e'. If there is no checking position along that path (if $\langle\downarrow\rangle(\mathcal{E}) = [\,]$), there is nowhere to insert a **let**. So instead, we give the lemma's "caller" the facts necessary to insert the **let** in a more shallow position (in whatever term has $\mathcal{E}[e']$ as a subterm). This explains part (2)(a). If we happen to have a checking position in $\mathcal{E}$ (as for example $\mathcal{E} = (x\,[\,])$), we can insert the **let** and produce a result (2)(b) that mirrors (1).

Proof. For (1), by induction on the derivation of $\Gamma; \Delta \vdash \mathcal{E}[e'] \downarrow C$; for (2), by induction on the derivation of $\Gamma; \Delta \vdash \mathcal{E}[e'] \uparrow C$. In order for the induction hypothesis to be well-founded in the case for rule sub, we consider a synthesis judgment $\Gamma; \Delta \vdash e \uparrow C_1$ to be smaller than a checking judgment $\Gamma; \Delta \vdash e \downarrow C_2$ (regardless of C_1 and C_2).

For (1), the cases fall into several categories:

(a) Cases that are impossible because there exists no evaluation context with a synthesizing subterm in evaluation position: $\mathbf{1}$I, $\rightarrow$I, fix.

(b) Cases where the term is duplicated: $\wedge$I, $\vee\mathbb{L}$, $\wedge$-ct. Here we apply the IH (1) to each premise. Since $\mathcal{E}'$ and $\mathcal{E}''$ are syntactically determined, they must be consistent across both applications of the IH. For $\wedge$I, it is clear that if $\mathcal{E}[e']$ value and $\mathcal{E} = \mathcal{E}'[\mathcal{E}'']$ then $\mathcal{E}'[\textbf{let}\ \overline{\mathbf{x}} = e'\ \textbf{in}\ \mathcal{E}''[\overline{\mathbf{x}}]]$ value.

(c) Cases with 0 or 1 premises, where the premise (if any) is in the checking direction and types the same term as the conclusion: $\top$I, ΠI, $\supset$I, ΣI, $\otimes$I, $\vee$I$_{1,2}$, contra, $\bot\mathbb{L}$, $\Sigma\mathbb{L}$, $\otimes\mathbb{L}$, $\wedge\mathbb{L}_{1,2}$, $\Pi\mathbb{L}$, $\supset\mathbb{L}$. Here we simply apply the IH (1) and reapply the rule, or, for $\top$I and contra, we reapply the rule to the appropriate term.

(d) Cases where all premises are in the checking direction and we decompose the subject term: $*$I, δI. In these cases we simply apply the IH (1) with a smaller evaluation context and reapply the rule. The syntactic conditions $\mathcal{E}' = \langle\Downarrow\rangle(\mathcal{E})$ and $\mathcal{E}' = \mathcal{E}'[\mathcal{E}'']$ are easy to satisfy using the result of applying the IH and the definition of $\langle\Downarrow\rangle(-)$.

(e) Other cases: let, sub, δE.

- **Case** let: $\mathcal{D} :: \dfrac{\Gamma;\Delta \vdash e_1 \uparrow B \qquad \Gamma;\Delta', \overline{\mathbf{y}}{:}B \vdash e_2 \downarrow C}{\Gamma;\Delta,\Delta' \vdash \textbf{let}\,\overline{\mathbf{y}} = e_1 \textbf{ in } e_2 \downarrow C}$

  We have $\mathcal{E}[e'] = (\textbf{let}\,\overline{\mathbf{y}} = e_1 \textbf{ in } e_2)$.

  e' must be a subterm of either e_1 or e_2.

  - e' is a subterm of e_1: Then $\mathcal{E} = (\textbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \textbf{ in } e_2)$ and $e_1 = \mathcal{E}_1[e']$ for some $\mathcal{E}_1$. By IH (2), either

    (a) there exists A' such that $\Gamma;\Delta_1 \vdash e' \uparrow A'$ and $\Gamma;\Delta_2, \overline{\mathbf{x}}{:}A' \vdash \mathcal{E}_1[\overline{\mathbf{x}}] \uparrow B$, or

    (b) $\mathcal{E}_1 \neq []$ and $\Gamma;\Delta \vdash \mathcal{E}_1'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } \mathcal{E}_1''[\overline{\mathbf{x}}]] \uparrow B$ where $\mathcal{E}_1'[\mathcal{E}_1''] = \mathcal{E}_1$ and $\mathcal{E}_1' = \langle\Downarrow\rangle(\mathcal{E}_1)$.

    Subcase (a):

    $\Delta = \Delta_1, \Delta_2$
    $\Gamma;\Delta_2, \overline{\mathbf{x}}{:}A' \vdash \mathcal{E}_1[\overline{\mathbf{x}}] \uparrow B$ — By IH (2)(a)
    $\Gamma;\Delta', \overline{\mathbf{y}}{:}B \vdash e_2 \downarrow C$ — Subd.
    $\Gamma;\Delta_2, \overline{\mathbf{x}}{:}A', \Delta' \vdash \textbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1[\overline{\mathbf{x}}] \textbf{ in } e_2 \downarrow C$ — By let

    $\Gamma;\Delta_1 \vdash e' \uparrow A'$ — By IH (2)(a)

    $\Gamma;\Delta_1, \Delta_2, \Delta' \vdash \textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in let}\,\overline{\mathbf{y}} = \mathcal{E}_1[\overline{\mathbf{x}}] \textbf{ in } e_2 \downarrow C$ — By let

    Let $\mathcal{E}' = []$
    and $\mathcal{E}'' = \mathcal{E} = \textbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \textbf{ in } e_2$.

    $\langle\Downarrow\rangle(\mathcal{E}_1) = []$ — By IH (2)(a)
    $\langle\Downarrow\rangle(\textbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \textbf{ in } e_2) = []$ — By defn. of $\langle\Downarrow\rangle(-)$

    ☞ $\mathcal{E}' = \langle\Downarrow\rangle(\mathcal{E})$
    $\mathcal{E} = \mathcal{E}'[\mathcal{E}]$ — By $\mathcal{E}' = []$
    ☞ $\mathcal{E} = \mathcal{E}'[\mathcal{E}'']$ — By $\mathcal{E}'' = \mathcal{E}$
    $\Gamma;\Delta_1, \Delta_2, \Delta' \vdash \textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in let}\,\overline{\mathbf{y}} = \mathcal{E}_1[\overline{\mathbf{x}}] \textbf{ in } e_2 \downarrow C$ — By let
    $\Gamma;\Delta_1, \Delta_2, \Delta' \vdash \mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } \mathcal{E}''[\overline{\mathbf{x}}]] \downarrow C$ — By $\mathcal{E}'' = \textbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \textbf{ in } e_2$ and $\mathcal{E}' = []$
    ☞ $\Gamma;\Delta, \Delta' \vdash \mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } \mathcal{E}''[\overline{\mathbf{x}}]] \downarrow C$ — By $\Delta = \Delta_1, \Delta_2$

    Subcase (b):

$$\begin{array}{rll}
\Gamma; \Delta \vdash & \mathcal{E}_1'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}_1''[\overline{\mathbf{x}}]] \uparrow B & \text{By IH (2)(b)} \\
\Gamma; \Delta', \overline{\mathbf{y}}{:}B \vdash & e_2 \downarrow C & \text{Subd.} \\
\Gamma; \Delta, \Delta' \vdash & \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}_1''[\overline{\mathbf{x}}]] \;\mathbf{in}\; e_2 \downarrow C & \text{By let}
\end{array}$$

Let $\mathcal{E}' = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1' \;\mathbf{in}\; e_2$ and $\mathcal{E}'' = \mathcal{E}_1''$.

☞ $\Gamma; \Delta, \Delta' \vdash \mathcal{E}'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}''[\overline{\mathbf{x}}]] \downarrow C$ — Using previous line

$$\begin{array}{rll}
\mathcal{E}_1 & \neq [\,] & \text{By IH (2)(b)} \\
\langle\Downarrow\rangle(\mathcal{E}) & = \langle\Downarrow\rangle(\mathcal{E}_1) & \text{By defn. of } \langle\Downarrow\rangle(-) \\
\mathcal{E}_1' & = \langle\Downarrow\rangle(\mathcal{E}_1) & \text{By IH (2)(b)}
\end{array}$$

$$\begin{array}{rll}
\mathcal{E}' & = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1' \;\mathbf{in}\; e_2 & \text{above} \\
& = \mathbf{let}\,\overline{\mathbf{y}} = \langle\Downarrow\rangle(\mathcal{E}_1) \;\mathbf{in}\; e_2 & \text{By } \mathcal{E}_1' = \langle\Downarrow\rangle(\mathcal{E}_1) \\
& = \langle\Downarrow\rangle(\mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \;\mathbf{in}\; e_2) & \text{By defn. of } \langle\Downarrow\rangle(-) \text{ and } \mathcal{E}_1 \neq [\,] \\
☞ & = \langle\Downarrow\rangle(\mathcal{E}) & \text{By } \mathcal{E} = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1' \;\mathbf{in}\; e_2
\end{array}$$

$$\begin{array}{rll}
\mathcal{E} & = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1 \;\mathbf{in}\; e_2 & \text{above} \\
& = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1'[\mathcal{E}_1''] \;\mathbf{in}\; e_2 & \text{By IH (2)(b)} \\
& = \mathcal{E}'[\mathcal{E}_1''] & \text{By } \mathcal{E}' = \mathbf{let}\,\overline{\mathbf{y}} = \mathcal{E}_1' \;\mathbf{in}\; e_2 \\
☞ & = \mathcal{E}'[\mathcal{E}''] & \text{By } \mathcal{E}'' = \mathcal{E}_1''
\end{array}$$

– e' is a subterm of e_2: In this case, there exists $\mathcal{E}_2$ such that $\mathcal{E} = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2$, and e_1 is a value.

$$\begin{array}{rll}
\Gamma; \Delta \vdash & e_1 \uparrow B & \text{Subd.} \\
\Gamma; \Delta', \overline{\mathbf{y}}{:}B \vdash & e_2 \downarrow C & \text{Subd.} \\
\Gamma; \Delta', \overline{\mathbf{y}}{:}B \vdash & \mathcal{E}_2'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}_2''[\overline{\mathbf{x}}]] \downarrow C & \text{By IH (1)} \\
\Gamma; \Delta, \Delta' \vdash & \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}_2''[\overline{\mathbf{x}}]] \downarrow C & \text{By let}
\end{array}$$

Let $\mathcal{E}' = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2'$ (allowed since e_1 value) and $\mathcal{E}'' = \mathcal{E}_2''$.

☞ $\Gamma; \Delta, \Delta' \vdash \mathcal{E}'[\mathbf{let}\,\overline{\mathbf{x}} = e' \;\mathbf{in}\; \mathcal{E}''[\overline{\mathbf{x}}]] \downarrow C$

$$\begin{array}{rll}
\mathcal{E}_2' & = \langle\Downarrow\rangle(\mathcal{E}_2) & \text{By IH(1)} \\
\mathcal{E}' & = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2' & \text{above} \\
& = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \langle\Downarrow\rangle(\mathcal{E}_2) & \text{By } \mathcal{E}_2' = \langle\Downarrow\rangle(\mathcal{E}_2) \\
& = \langle\Downarrow\rangle(\mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2) & \text{By defn. of } \langle\Downarrow\rangle(-) \\
☞ & = \langle\Downarrow\rangle(\mathcal{E}) & \text{By } \mathcal{E} = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; E_2
\end{array}$$

$$\begin{array}{rll}
\mathcal{E}_2 & = \mathcal{E}_2'[\mathcal{E}_2''] & \text{By IH(1)} \\
\mathcal{E} & = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2 & \text{above} \\
& = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2'[\mathcal{E}_2''] & \text{By } \mathcal{E}_2 = \mathcal{E}_2'[\mathcal{E}_2''] \\
☞ & = \mathcal{E}'[\mathcal{E}''] & \text{By } \mathcal{E}' = \mathbf{let}\,\overline{\mathbf{y}} = e_1 \;\mathbf{in}\; \mathcal{E}_2' \text{ and } \mathcal{E}'' = \mathcal{E}_2''
\end{array}$$

- **Case** sub: Similar to the first subcase of let: apply IH (2) and either:

 - apply sub to $\mathcal{E}[\overline{\mathbf{x}}]$ then apply let, or
 - apply sub to $\mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]]$.

- **Case** δE: $\mathcal{E}$ must be **case** $\mathcal{E}_0$ **of** ms for some $\mathcal{E}_0$. Similar to sub.

Part (2):
Either $\mathcal{E} = []$ or $\mathcal{E} \neq []$.
If $\mathcal{E} = []$, we show (2)(a): Let $A' = C$ and $\Delta_2 = \cdot$. Then $\Gamma; \Delta \vdash \mathcal{E}[e'] \uparrow C$ leads to $\Gamma; \Delta_1 \vdash e' \uparrow A'$. By $\overline{\mathsf{var}}$, $\Gamma; \Delta_2, \overline{\mathbf{x}}{:}A' \vdash \mathcal{E}[\overline{\mathbf{x}}] \uparrow C$.
If $\mathcal{E} \neq []$, we show (2)(b), and distinguish the following cases:

- **Cases** ΠE, $\wedge$E$_1$, $\wedge$E$_2$, $\supset$E: Apply the IH (2). Since $\mathcal{E} \neq []$, we obtain (2)(b). Reapplying the rule yields the result.
- **Case** ctx-anno: IH (1) then show (2)(b).
- **Cases** var, fixvar: Under the assumption that $\mathcal{E} \neq []$, these cases are impossible.
- **Case** $\rightarrow$E: If e' is a subterm of e_1, use IH (2), then reapply $\rightarrow$E, showing (2)(b). If e' is a subterm of e_2, use IH (1) then reapply $\rightarrow$E, showing (2)(b).
- **Case** $\overline{\mathsf{var}}$: Impossible: e' cannot be a linear variable. □

Corollary 5.49 (Letification). *If $e = \mathcal{C}[e']$ where e' is in synthesizing form and is not a linear variable and is not the right hand side of a let-binding, and either*

(1) $\Gamma; \Delta \vdash e \downarrow C$*, or*

(2) $\Gamma; \Delta \vdash e \uparrow C$ *and* $\mathcal{C}$ *is* not *an evaluation context in* e

then $\Gamma; \Delta \vdash \mathcal{C}'[\mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]]] \downarrow C$ *(if (1)),*
or $\Gamma; \Delta \vdash \mathcal{C}'[\mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]]] \uparrow C$ *(if (2))*
for some $\mathcal{C}'$, $\mathcal{E}'$, $\mathcal{E}''$ *such that* $e = \mathcal{C}'[\mathcal{E}'[\mathcal{E}''[e']]]$.

Proof. By induction on the given derivation. For case (1):

- If e' is in evaluation position in e, use Lemma 5.47.
- Otherwise, e' is not in evaluation position in e. If a "short-circuit" rule—contra or $\bot\mathbb{L}$—concludes the derivation, apply that rule again to $\mathcal{C}[\mathcal{E}'[\textbf{let}\,\overline{\mathbf{x}} = e'\,\textbf{in}\,\mathcal{E}''[\overline{\mathbf{x}}]]]$ for some $\mathcal{E}'$, $\mathcal{E}''$ such that $e_0 = \mathcal{E}'[\mathcal{E}''[e']]$ is a subterm of e. Use the IH on the premise whose subject has e' as a subterm. If that premise is synthesizing, its subject is in evaluation position in e (for example, in δE on $e = \textbf{case}\ e_1\ \textbf{of}\ ms$, e_1 is in evaluation position)[7]. Suppose e' were in evaluation position in the premise's subject; then it would be in evaluation position in e, a contradiction. Hence the side condition on (2) is satisfied.

  We finish by reapplying the rule.

[7] Every rule in the type system has its premises ordered left-to-right by subject: in $\rightarrow$E, typing $e_1\,e_2$, the first premise's subject is e_1 and the second premise's subject is e_2. Every set of premises so ordered consists of zero or one synthesis judgments, *followed by* zero or more checking judgments. Thus the synthesizing premise, if any, "hugs the left edge" of the term.

For case (2), e' cannot be in evaluation position in e. Short-circuit rules are impossible (they all have a checking judgment as conclusion), but otherwise proceed as in the second bullet in case (1) above. □

Slackening

Here we show that slackening a variable $\bar{\mathbf{x}}$—changing $\mathbf{let}\,\bar{\mathbf{x}} = v\ \mathbf{in}\ e$ to $\mathbf{let}\ {\sim}\bar{\mathbf{x}} = v\ \mathbf{in}\ e$—preserves typing. We must slacken $\bar{\mathbf{x}}$ when v has the form $(v' : As)$; it would be pointless to do it for other v, since all other v have the principal synthesis property and need not be slackened.

Lemma 5.50. *If* $\Gamma; \Delta_1 \vdash v \uparrow A$ *and* $\Gamma; \Delta_2, \bar{\mathbf{x}}{:}A \vdash e \downarrow C$
then $\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ {\sim}\bar{\mathbf{x}} = v\ \mathbf{in}\ e \downarrow C$.

Proof. By ${\sim}\mathsf{var}$, $\Gamma; \Delta_1, \Delta_2, {\sim}\bar{\mathbf{x}} = v \vdash e \downarrow C$. By $\mathsf{let}{\sim}$, $\Gamma; \Delta_1, \Delta_2 \vdash \mathbf{let}\ {\sim}\bar{\mathbf{x}} = v\ \mathbf{in}\ e \downarrow C$. □

Corollary 5.51 (Slackening)**.** *If* $\Gamma; \Delta \vdash \mathcal{C}[\mathbf{let}\,\bar{\mathbf{x}} = v_1\ \mathbf{in}\ e_2] \downarrow C$ *then* $\Gamma; \Delta \vdash \mathcal{C}[\mathbf{let}\ {\sim}\bar{\mathbf{x}} = v_1\ \mathbf{in}\ e_2] \downarrow C$.

Proof. By induction on the given derivation. In the case when a short-circuit rule such as contra was used we simply replace the **let** with $\mathbf{let}\ {\sim}\bar{\mathbf{x}} = v\ \mathbf{in}\ e$ and reapply. If let was used and $\mathcal{C} = [\,]$, we apply Lemma 5.50, yielding the desired result. In all other cases, we apply the IH to the derivation whose subject contains the **let** and reapply the rule. □

Permutation

In this section, we prove that moving a **let** or **let**~ leftward in a term, passing over no other **let**s, preserves typing:

$\mathcal{W}$: a let-free viable path, Def. 5.32

$$\text{If } \Gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\,[\sim]\,\bar{\mathbf{x}} = e'\ \mathbf{in}\ e] \downarrow C \text{ then } \Gamma; \Delta \vdash \mathbf{let}\,[\sim]\,\bar{\mathbf{x}} = e'\ \mathbf{in}\ \mathcal{W}[e] \downarrow C.$$

For **let**~ bindings, this can be proved quite easily. Recall that for $\mathbf{let}\ {\sim}\bar{\mathbf{x}} = e'\ \mathbf{in}\ e$, the variable $\bar{\mathbf{x}}$ is bound to e' without examining e'; synthesizing a type for e' can be done at any later point. Thus, we can move a slack binding backward without changing the part of the derivation where we synthesize the type of e'.

Lemma 5.52 (Slack Binding Inversion)**.** *If* $\Gamma; \Delta \vdash \mathbf{let}\ {\sim}\bar{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C$ *then* $\Gamma; \Delta, {\sim}\bar{\mathbf{x}} = v_2 \vdash e_3 \downarrow C$.

Proof. By induction on the given derivation.

- **Case** $\mathsf{let}{\sim}$: The subderivation constitutes the result.
- **Case** $\bot\mathbb{L}$: Apply $\bot\mathbb{L}$ to yield $\Gamma; \Delta, {\sim}\bar{\mathbf{x}} = v_2 \vdash e_3 \downarrow C$.
- **Case** $\vee\mathbb{L}$: Apply IH to each premise, then reapply $\vee\mathbb{L}$.
- **Cases** $\Sigma\mathbb{L}$, $\otimes\mathbb{L}$, $\wedge\mathbb{L}_1$, $\wedge\mathbb{L}_2$, $\Pi\mathbb{L}$, $\supset\mathbb{L}$: Apply IH to the premise, then reapply.
- **Case** sub: Impossible since $\mathbf{let}\ {\sim}\bar{\mathbf{x}} = v_2\ \mathbf{in}\ e_3$ cannot synthesize a type.

- **Case** contra: Apply contra.
- **Case** $\top$I, $\wedge$I, ΠI: Given ($\mathbf{let} \sim\overline{\mathbf{x}} = v_2 \mathbf{~in~} e_3$) value, we have e_3 value. Apply the IH to each premise (if any), then reapply the rule.
- **Cases** $\vee$I$_1$, $\vee$I$_2$, ΣI: Apply the IH to the premise, then reapply the rule. □

The $\underline{\downarrow\uparrow}$ notation was first used in Lemma 3.13 and is described on page 68.

Lemma 5.53. *If* $\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let} \sim\overline{\mathbf{x}} = e' \mathbf{~in~} e] \;\underline{\downarrow\uparrow}\; C$ *then* $\Gamma; \Delta, \sim\overline{\mathbf{x}} = e' \vdash \mathcal{W}[e] \;\underline{\downarrow\uparrow}\; C$.

Proof. By induction on the given derivation. If $\mathcal{W} = \cdot$ use Lemma 5.52. Otherwise apply the IH to the appropriate premise and reapply the rule. □

Lemma 5.54. *If* $\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let} \sim\overline{\mathbf{x}} = e' \mathbf{~in~} e] \downarrow C$ *then* $\Gamma; \Delta \vdash \mathbf{let} \sim\overline{\mathbf{x}} = e' \mathbf{~in~} \mathcal{W}[e] \downarrow C$.

Proof. By Lemma 5.53, $\Gamma; \Delta, \sim\overline{\mathbf{x}} = e' \vdash \mathcal{W}[e] \downarrow C$. Applying let$\sim$ gives the result. □

The judgment $\{(\Gamma_1; \Delta_1), \ldots, (\Gamma_n; \Delta_n)\} \;\boxed{\Psi}\; \Gamma'; \Delta'$, read "the set $\Gamma_1; \Delta_1$ through $\Gamma_n; \Delta_n$ joins $\Gamma'; \Delta'$" (in a way corresponding to the left rules) is defined in Figure 5.9.

Proposition 5.55. *If* $\{(\Gamma_1; \Delta_1), \ldots, (\Gamma_n; \Delta_n)\} \;\boxed{\Psi}\; \Gamma; \Delta$, *and*
$\Gamma_1; \Delta_1 \vdash e \downarrow B$
and $\vdots$
and $\Gamma_n; \Delta_n \vdash e \downarrow B$
are derivable, $\Gamma; \Delta \vdash e \downarrow B$ *is derivable.*

Proof. By induction on the given derivation. □

Lemma 5.56. *If* $\{\Gamma_i; \Delta_i\} \;\boxed{\Psi}\; \Gamma'; \Delta'$ *then* $\{\Gamma_i; \Delta_i, \Delta\} \;\boxed{\Psi}\; \Gamma'; \Delta', \Delta$.

Proof. By induction on the given derivation. □

Proposition 5.57. *If* $\Gamma \vdash A \uparrow B$ *then* $\{\Gamma; \Delta, \overline{\mathbf{x}}{:}B\} \;\boxed{\Psi}\; \Gamma; \Delta, \overline{\mathbf{x}}{:}A$.

Proof. By induction on $\Gamma \vdash A \uparrow B$. □

Lemma 5.58. *If* $\Gamma \vdash A' \uparrow A$ *then* $\Gamma; \overline{\mathbf{x}}{:}A' \vdash \overline{\mathbf{x}} \uparrow A$.

Proof. By induction on the derivation of $\Gamma \vdash A' \uparrow A$. □

Lemma 5.59 (Synthesis Subtyping Weakening)**.** *If* $\Gamma; \Delta_1, \overline{\mathbf{x}}{:}A, \Delta_2 \vdash e \;\underline{\downarrow\uparrow}\; B$ *and* $\Gamma \vdash A' \uparrow A$ *then* $\Gamma; \Delta_1, \overline{\mathbf{x}}{:}A', \Delta_2 \vdash e \;\underline{\downarrow\uparrow}\; B$.

Proof. By induction on the first derivation. In most cases, simply apply the IH and reapply the rule. In the $\overline{\text{var}}$ case, use Lemma 5.58. □

The result we would now like to obtain is the following, which moves a binding outward over $\mathcal{W}$:

$$\boxed{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta} \qquad \Gamma\Delta s \;::=\; \cdot \mid \Gamma\Delta s, (\Gamma;\Delta)$$

$$\frac{}{\{(\Gamma;\Delta)\} \;\boxed{\Psi}\; \Gamma;\Delta}\ \Psi\text{-refl}$$

$$\frac{\Gamma;\Delta,\bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}A \wedge B \vdash e \downarrow C}\ \wedge\mathbb{L}_1 \qquad \frac{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A \wedge B}\ \wedge\Psi_1$$

$$\frac{\Gamma;\Delta,\bar{\mathbf{x}}{:}B \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}A \wedge B \vdash e \downarrow C}\ \wedge\mathbb{L}_2 \qquad \frac{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}B}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A \wedge B}\ \wedge\Psi_2$$

$$\frac{\Gamma;\Delta,\bar{\mathbf{x}}{:}[i/a]A \vdash e \downarrow C \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma;\Delta,\bar{\mathbf{x}}{:}\Pi a{:}\gamma.\,A \vdash e \downarrow C}\ \Pi\mathbb{L} \qquad \frac{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}[i/a]A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}\Pi a{:}\gamma.\,A}\ \Pi\Psi$$

$$\frac{\Gamma, a{:}\gamma;\Delta,\bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}\Sigma a{:}\gamma.\,A \vdash e \downarrow C}\ \Sigma\mathbb{L} \qquad \frac{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma, a{:}\gamma;\Delta,\bar{\mathbf{x}}{:}A}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}\Sigma a{:}\gamma.\,A}\ \Sigma\Psi$$

$$\frac{\overline{\Gamma} \models P \qquad \Gamma;\Delta,\bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}P \supset A \vdash e \downarrow C}\ \supset\mathbb{L} \qquad \frac{\overline{\Gamma} \models P \qquad \Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}P \supset A}\ \supset\Psi$$

$$\frac{\Gamma, P;\Delta,\bar{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}(P \otimes A) \vdash e \downarrow C}\ \otimes\mathbb{L} \qquad \frac{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma, P, a{:}\gamma;\Delta,\bar{\mathbf{x}}{:}A}{\Gamma\Delta s \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}(P \otimes A)}\ \otimes\Psi$$

$$\frac{\Gamma;\Delta,\bar{\mathbf{x}}{:}A \vdash e \downarrow C \qquad \Gamma;\Delta,\bar{\mathbf{x}}{:}B \vdash e \downarrow C}{\Gamma;\Delta,\bar{\mathbf{x}}{:}A \vee B \vdash e \downarrow C}\ \vee\mathbb{L} \qquad \frac{\Gamma\Delta s_1 \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A \qquad \Gamma\Delta s_2 \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}B}{\Gamma\Delta s_1 \cup \Gamma\Delta s_2 \;\boxed{\Psi}\; \Gamma;\Delta,\bar{\mathbf{x}}{:}A \vee B}\ \vee\Psi$$

Figure 5.9: Definition of the 'joins' judgment. The left rules are repeated in this figure to show their relationship to the 'joins' rules.

If $\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow C$ *then* $\Gamma; \Delta \vdash \mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e] \downarrow C$.

However, we cannot prove that without the lemma below. We need a synthesis version of the above (or we cannot apply the IH to a $\uparrow$-premise, such as in rule sub), but we cannot just flip the $\downarrow$s to $\uparrow$s because $\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e]$ cannot synthesize anything. So instead the synthesis version must show that e' synthesizes something and $\mathcal{W}[e]$ synthesizes something. Unfortunately, that is not enough: if $\vee\mathbb{L}$ analyzes a linear variable free in e', after pushing e' outward we cannot reapply $\vee\mathbb{L}$ with subject $\mathcal{W}[e]$ because linear variables free in e' are no longer available in the linear context. Nor can we reapply $\vee\mathbb{L}$ with subject e', because we need $e' \uparrow \ldots$, not $e' \downarrow \ldots$, and $\vee\mathbb{L}$ is a checking rule. We must delay applying $\vee\mathbb{L}$ until the checking mode is entered. Thus we produce not just a single $e' \uparrow \ldots$ and $\mathcal{W}[e] \uparrow \ldots$ but a set of them, with varying linear contexts (as the linear contexts in the premises of $\vee\mathbb{L}$ differ). But we know that those varying linear contexts can be reconciled by applying $\vee\mathbb{L}$ again; we express that knowledge through the $\boxed{\Psi}$ judgment.

So much for the inductive reasoning when we go from a checking judgment in a rule's conclusion to a synthesis judgment in a premise; but we also have rules like $\to$E. To usefully apply the induction hypothesis if $\mathcal{W} = v\mathcal{W}'$ we must yield a set of $e' \uparrow \ldots$ and $\mathcal{W}[e] \downarrow \ldots$ judgments with varying linear contexts in the checking version of the result, not just in the synthesis version. Finally, we have formulated part (b) of the lemma below.

The very last wrinkle is that in the contra and $\bot\mathbb{L}$ cases, we cannot show $e' \uparrow \ldots$ at all. However, both contra and $\bot\mathbb{L}$ are based on Γ and Δ only, so $\Gamma; \Delta \vdash e^* \downarrow D$ for *any* e^* and D. This is part (a). At last, we know what to prove:

Lemma 5.60 (Let Inversion). *If* $\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e] \,\underline{\downarrow\uparrow}\, C$ *and* $\Delta \Vdash \mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e]$ ok *where* e' *does not have the form* $(v : As)$[8] *and does not have the form* $\overline{\mathbf{y}}$ *then either*

(a) for all D, Γ^*, Δ^* *such that* $\Gamma, \Gamma^* \vdash e^*$ ok *and* $\Delta, \Delta^* \Vdash e^*$ ok,
it is the case that $\Gamma, \Gamma^*; \Delta, \Delta^* \vdash e^* \downarrow D$, *or*

(b) for i *from* 1 *to* n, *there exist* Γ_i, A_i, Δ_{i1}, Δ_{i2} *such that*

$$\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i \quad \textit{and} \quad \Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \,\underline{\downarrow\uparrow}\, C$$

and $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \;\boxed{\Psi}\; \Gamma; \Delta$.

Remark 5.61. The requirement $\Delta \Vdash \mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e]$ ok excludes terms like $\mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = \overline{\mathbf{y}}() \,\mathbf{in}\, e]$ where $\mathcal{W}$ contains a binding of $\overline{\mathbf{y}}$ and $\overline{\mathbf{y}} \notin \mathrm{dom}(\Delta)$.

Proof. By induction on the first derivation.

- Cases var, $\to$I, fixvar, fix, **1**I are impossible, because the term typed cannot possibly have the form $\mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e]$.

- **Case** $\to$E: $\mathcal{D} :: \dfrac{\Gamma; \Delta_1 \vdash \mathcal{W}'[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e] \uparrow B \to C \qquad \Gamma; \Delta_2 \vdash e_2 \downarrow B}{\Gamma; \Delta_1, \Delta_2 \vdash (\mathcal{W}'[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e])\, e_2 \uparrow C}$

[8]When this lemma is invoked, all $(v : As)$ bindings will have been slackened with Lemma 5.54.

(This is the first $\to$E subcase, in which the **let** appears inside the function expression: $\mathcal{W} = \mathcal{W}' e_2$.)

It is given that $\Delta \Vdash \textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } \mathcal{W}[e]$ ok. Since $\Delta = \Delta_1, \Delta_2$ and $\mathcal{W} = \mathcal{W}' e_2$,

$$\Delta_1, \Delta_2 \Vdash \textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } (\mathcal{W}'[e])\, e_2 \text{ ok}$$

By Proposition 5.12, $\Delta_2 \Vdash e_2$ ok. It is then clear from the definition of $\Vdash$ that we can strip Δ_2 and e_2 out to get

$$\Delta_1 \Vdash \textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } \mathcal{W}'[e] \text{ ok}$$

which lets us apply the IH to the first premise. If it shows part (a), we are done (note that part (a) is not tied to a particular term).

If it shows (b), then for each i, let $\Delta_{i1} = \Delta'_{i1}$ and $\Delta_{i2} = \Delta_1, \Delta'_{i2}$.

☞	$\Gamma, \Gamma_i; \Delta'_{i1} \vdash e' \uparrow A_i$	From IH
	$\Gamma, \Gamma_i; \Delta'_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}'[e] \uparrow B \to C$	From IH
	$\Gamma; \Delta_2 \vdash e_2 \downarrow B$	Subd.
	$\Gamma, \Gamma_i; \Delta_2 \vdash e_2 \downarrow B$	Weakening
	$\Gamma, \Gamma_i; \Delta'_{i2}, \Delta_2, \overline{\mathbf{x}}{:}A_i \vdash (\mathcal{W}'[e])\, e_2 \uparrow C$	By $\to$E
☞	$\Gamma, \Gamma_i; \Delta'_{i2}, \Delta_2, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \uparrow C$	By $\mathcal{W} = \mathcal{W}' e$

To show the joining:

	$\Gamma, \Gamma_i; \Delta'_{i1}, \Delta'_{i2}\ \boxed{\Psi}\ \Gamma; \Delta_1$	From IH
☞	$\Gamma, \Gamma_i; \Delta'_{i1}, \Delta'_{i2}, \Delta_2\ \boxed{\Psi}\ \Gamma; \Delta_1, \Delta_2$	By Lemma 5.56

- **Case** $\to$E: $\boxed{\mathcal{D} :: \dfrac{\Gamma; \Delta_1 \vdash e_1 \uparrow B \to C \qquad \Gamma; \Delta_2 \vdash \mathcal{W}'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow B}{\Gamma; \Delta_1, \Delta_2 \vdash e_1(\mathcal{W}'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } e]) \uparrow C}}$

(This is the second $\to$E subcase, in which the **let** appears inside the argument expression: $\mathcal{W} = e_1\, \mathcal{W}'$.)

Apply the IH to the subderivation $\Gamma; \Delta_2 \vdash \mathcal{W}'[\textbf{let}\,\overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow B$. If it shows part (a), we are done (part (a) is tied only to the context Γ, which is the same in the conclusion and second premise of $\to$E).

If it shows (b), then for each i, let $\Delta_{i1} = \Delta_1, \Delta'_{i1}$ and $\Delta_{i2} = \Delta'_{i2}$.

☞	$\Gamma, \Gamma_i; \Delta'_{i1} \vdash e' \uparrow A_i$	From IH
	$\Gamma, \Gamma_i; \Delta'_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}'[e] \downarrow B$	From IH
	$\Gamma; \Delta_1 \vdash e_1 \uparrow B \to C$	Subd.
	$\Gamma, \Gamma_i; \Delta_1 \vdash e_1 \uparrow B \to C$	Weakening
	$\Gamma, \Gamma_i; \Delta_1, \Delta'_{i2}, \overline{\mathbf{x}}{:}A_i \vdash e_1\, (\mathcal{W}'[e]) \uparrow C$	By $\to$E
☞	$\Gamma, \Gamma_i; \Delta_1, \Delta'_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \uparrow C$	By $\mathcal{W} = e_1 \mathcal{W}'$

To show the joining:

$\Gamma, \Gamma_i; \Delta'_{i1}, \Delta'_{i2} \boxed{\Psi} \Gamma; \Delta_2$ From IH

☞ $\Gamma, \Gamma_i; \Delta_1, \Delta'_{i1}, \Delta'_{i2} \boxed{\Psi} \Gamma; \Delta_1, \Delta_2$ By Lemma 5.56

- **Case** sub: Apply IH to show that either (a) or (b) holds. If (a), we're done (part (a) is dependent only on Γ and Δ, which are the same in the premise and conclusion of sub). If (b):

  ☞ $\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i$ (for $i \in 1..n$) IH (b)

  ☞ $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \boxed{\Psi} \Gamma; \Delta$ IH (b)

  $\Gamma, \Gamma_i; \Delta_{i2}; \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \uparrow C$ (for $i \in 1..n$) IH (b)

  $\Gamma, \Gamma_i \vdash C \leq C$ (for $i \in 1..n$) By Lemma 2.9

  ☞ $\Gamma, \Gamma_i; \Delta_{i2}; \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C$ (for $i \in 1..n$) By sub

- **Cases** $*$I, δI, δE, $*E_1$, $*E_2$, $\wedge E_{1,2}$, ΠE, ctx-anno: Roughly analogous to case $\to$E.

- **Case** let: $\mathcal{D} :: \dfrac{\Gamma; \Delta' \vdash e_1 \uparrow B \qquad \Gamma; \Delta'', \overline{\mathbf{y}}{:}B \vdash e_2 \downarrow C}{\Gamma; \Delta', \Delta'' \vdash \textbf{let}\, \overline{\mathbf{y}} = e_1 \,\textbf{in}\, e_2 \downarrow C}$

  If $\mathcal{W} = []$ then the subderivations provide what we needed to show $(\{(\Gamma; \Delta', \Delta'')\} \boxed{\Psi} (\Gamma; \Delta', \Delta'')$ by Ψ-refl). Otherwise, by definition of $\mathcal{W}$, $\mathcal{W} = \textbf{let}\, \overline{\mathbf{y}} = \mathcal{W}'[\textbf{let}\, \overline{\mathbf{x}} = e' \,\textbf{in}\, e] \,\textbf{in}\, e_2$.

  $e_1 = \mathcal{W}'[\textbf{let}\, \overline{\mathbf{x}} = e' \,\textbf{in}\, e]$ Given

  $\Gamma; \Delta' \vdash \mathcal{W}'[\textbf{let}\, \overline{\mathbf{x}} = e' \,\textbf{in}\, e] \uparrow B$ Subd.

  Use the IH. If (a), we're done. If (b), then

  ☞ $\{(\Gamma; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \boxed{\Psi} \Gamma; \Delta', \Delta''$ By IH (b)

  and for all i:

  ☞ $\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i$ By IH (b)

  $\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}'[e] \uparrow B$ By IH (b)

  $\Gamma; \Delta'', \overline{\mathbf{y}}{:}B \vdash e_2 \downarrow C$ Subd.

  $\Gamma, \Gamma_i; \Delta'', \overline{\mathbf{y}}{:}B \vdash e_2 \downarrow C$ Weakening

  ☞ $\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i, \Delta'' \vdash \textbf{let}\, \overline{\mathbf{y}} = \mathcal{W}'[e] \,\textbf{in}\, e_2 \downarrow C$ By let

- **Case** ΣI: $\mathcal{D} :: \dfrac{\Gamma; \Delta \vdash \mathcal{W}[\textbf{let}\, \overline{\mathbf{x}} = e' \,\textbf{in}\, e] \downarrow [j/a]\, C' \qquad \overline{\Gamma} \vdash j : \gamma}{\Gamma; \Delta \vdash \mathcal{W}[\textbf{let}\, \overline{\mathbf{x}} = e' \,\textbf{in}\, e] \downarrow \Sigma a{:}\gamma.\, C'}$

  Use the IH. If (a), we're done. If (b):

  $\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i$ and $\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow [i/a]\, C'$ (for $i \in 1..n$) From IH

  $\overline{\Gamma} \vdash j : \gamma$ Subd.

  $\overline{\Gamma, \Gamma_i} \vdash j : \gamma$ (for $i \in 1..n$) By Property 2.3

  ☞ $\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow \Sigma a{:}\gamma.\, C'$ (for $i \in 1..n$) By ΣI

  ☞ $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \boxed{\Psi} \Gamma; \Delta$ From IH

- **Case** contra: Show (a).

- **Case** $\top$I: Let $\Delta_{11}, \Delta_{12} = \Delta$. We have a term $\mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = v' \,\mathbf{in}\, v_2]$ value that checks against C. By assumption, v' is not an annotation. Therefore we can apply Lemma 5.4 to show there exists A_1 such that $\Gamma; \Delta_{11} \vdash v' \uparrow A_1$. By $\top$I, $\Gamma; \Delta_{12} \vdash \mathcal{W}[v_2] \downarrow \top$.

- **Case** ΠI: $\mathcal{D} :: \dfrac{\Gamma, a{:}\gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow C'}{\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow \Pi a{:}\gamma.\, C'}$

  We have $\mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e]$ value, so e' value, e value, and $\mathcal{W}[e]$ value.

  If the IH yields (a): We cannot "short out" instantly as usual, since $\Gamma, a{:}\gamma$ is not the same as Γ; we must show (b). We know e' is a value and that it is a synthesizing form. The only syntactic forms that are synthesizing values are annotations, linear variables, and ordinary variables. By assumption, e' is neither an annotation nor a linear variable. Therefore it must be some ordinary variable y. By Lemma 5.4, $\Gamma; \cdot \vdash e' \uparrow A^*$ for some A^*. Now let $e^* = \mathcal{W}[e]$ and $D = C'$; from the IH, we get $\Gamma, a{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C'$. By ΠI, $\Gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow \Pi a{:}\gamma.\, C'$. Since $\{\Gamma; \Delta\}\ \boxed{\Psi}\ \Gamma; \Delta$, we're done.

  If the IH yields (b):

  $\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i1}' \vdash e' \uparrow A_i$ and $\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i2}', \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C'$ (for $i \in 1..n$) From IH
  $\{\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i1}', \Delta_{i2}' \mid i \in 1..n\}\ \boxed{\Psi}\ \Gamma, a{:}\gamma; \Delta$ From IH
  $\Gamma, \Gamma_i'; \Delta_{i2}', \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow \Pi a{:}\gamma.\, C'$ (for $i \in 1..n$) By ΠI

  Given $\Gamma, a{:}\gamma, \Gamma_i'; \cdot \vdash e' \uparrow A_i$ for all i, we have by principal synthesis (Lemma 5.3) that there exists A^* such that $\Gamma, a{:}\gamma, \Gamma_i'; \cdot \vdash e' \uparrow A^*$ and for all i, $\Gamma, a{:}\gamma, \Gamma_i' \vdash A^* \uparrow A_i$. In fact, $A^* = \Gamma(x)$, and $\{a\} \cup FV(\Gamma_i')$ and $FV(A^*)$ are disjoint, so $\Gamma; \cdot \vdash e' \uparrow A^*$.

  Recall that the multiplicity of derivations resulted from the subcase of $\vee\mathbb{L}$ in which the linear variable of union type being split by $\vee\mathbb{L}$ appeared in e'. In the present situation (resulting from the value restriction), e' is some variable y, so we can join the judgments now.

  Let $\Delta_{12} = \Delta$ and $\Gamma_1 = \cdot$ and $\Delta_{11} = \cdot$.

  $\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i2}', \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C'$ (for $i \in 1..n$) Above
  $\Gamma, a{:}\gamma, \Gamma_i' \vdash A^* \uparrow A_i$ (for $i \in 1..n$) Above
  $\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i2}', \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C'$ (for $i \in 1..n$) By Lemma 5.59

  $\{\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i2}' \mid i \in 1..n\}\ \boxed{\Psi}\ \Gamma, a{:}\gamma; \Delta$ Above
  $\{\Gamma, a{:}\gamma, \Gamma_i'; \Delta_{i2}', \overline{\mathbf{x}}{:}A^* \mid i \in 1..n\}\ \boxed{\Psi}\ \Gamma, a{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A^*$ By Lemma 5.56
  $\Gamma, a{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C'$ By Proposition 5.55
  $\mathcal{W}[\mathbf{let}\,\overline{\mathbf{x}} = e' \,\mathbf{in}\, e]$ value ΠI restricted to values
  $\mathcal{W}[e]$ value
  $\Gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow \Pi a{:}\gamma.\, C'$ By ΠI
  ☞ $\Gamma, \Gamma_1; \Delta_{12}, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow \Pi a{:}\gamma.\, C'$ By $\Delta_{12} = \Delta$ and $\Gamma_1 = \cdot$

$\Gamma; \cdot \vdash y \uparrow A^*$ Above

☞ $\Gamma, \Gamma_1; \Delta_{11} \vdash y \uparrow A^*$ By $\Gamma_1 = \cdot$ and $\Delta_{11} = \cdot$

$\{\Gamma; \Delta\} \boxed{\Psi} \Gamma; \Delta$ By Ψ-refl

☞ $\{\Gamma, \Gamma_1; \Delta_{12}\} \boxed{\Psi} \Gamma; \Delta$ By $\Delta_{12} = \Delta$ and $\Gamma_1 = \cdot$

- **Case** $\supset$I: $\boxed{\mathcal{D} :: \dfrac{\Gamma, P; \Delta \vdash \mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow C'}{\Gamma; \Delta \vdash \mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow P \supset C'}}$

  Use the IH. If (a), we're done. If (b):

  $\left.\begin{array}{r} \Gamma, P, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i \\ \Gamma, P, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C' \end{array}\right\}$ (for $i \in 1..n$) By IH

  $\mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e]$ value $\supset$I restricted to values

  $\mathcal{W}[e]$ value

  ☞ $\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow P \supset C'$ (for $i \in 1..n$) By $\supset$I

  ☞ $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \boxed{\Psi} \Gamma; \Delta$ By IH

- **Case** $\wedge$I: $\boxed{\mathcal{D} :: \dfrac{\Gamma; \Delta \vdash \mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow C_1 \qquad \Gamma; \Delta \vdash \mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow C_2}{\Gamma; \Delta \vdash \mathcal{W}[\textbf{let } \overline{\mathbf{x}} = e' \textbf{ in } e] \downarrow C_1 \wedge C_2}}$

  Use the IH twice.

  – If either use of the IH shows (a), show (a).

  – If both uses show (b), much of the reasoning follows the ΠI case above; we omit some of the detail here. As in ΠI we have a value restriction; by assumption, e' is not an annotation; we assume that there is no binding with just a linear variable on the right hand side. Thus, e' must be some ordinary variable y.

IH on first premise	IH on second premise
$\{(\Gamma, \Gamma_i^1; \cdot, \Delta_{i2}^1) \mid i \in 1..n_1\} \boxed{\Psi} \Gamma; \Delta$	$\{(\Gamma, \Gamma_i^2; \cdot, \Delta_{i2}^2) \mid i \in 1..n_2\} \boxed{\Psi} \Gamma; \Delta$
$\Gamma, \Gamma_i^1; \cdot \vdash y \uparrow A_i^1$ for $i \in 1..n_1$	$\Gamma, \Gamma_i^2; \cdot \vdash y \uparrow A_i^2$ for $i \in 1..n_2$
$\Gamma, \Gamma_i^1; \Delta_{i2}^1, \overline{\mathbf{x}}{:}A_i^1 \vdash \mathcal{W}[e] \downarrow C_1$ for $i \in 1..n_1$	$\Gamma, \Gamma_i^2; \Delta_{i2}^2, \overline{\mathbf{x}}{:}A_i^2 \vdash \mathcal{W}[e] \downarrow C_2$ for $i \in 1..n_2$

  By Lemma 5.3, $\Gamma; \cdot \vdash y \uparrow A^*$ such that for all $i \in 1..n_1$, $A^* \uparrow A_i^1$, and for $i \in 1..n_2$, $A^* \uparrow A_i^2$.

  $\{(\Gamma, \Gamma_i^1; \cdot, \Delta_{i2}^1) \mid i \in 1..n_1\} \boxed{\Psi} \Gamma; \Delta\}$ Above

  $\{(\Gamma, \Gamma_i^1; \cdot, \Delta_{i2}^1, \overline{\mathbf{x}}{:}A^*) \mid i \in 1..n_1\} \boxed{\Psi} \Gamma; \Delta, \overline{\mathbf{x}}{:}A^*\}$ By Lemma 5.56

  $\Gamma, \Gamma_i^1; \Delta_{i2}^1, \overline{\mathbf{x}}{:}A_i^1 \vdash \mathcal{W}[e] \downarrow C_1$ (for all $i \in 1..n_1$) Above

  $\Gamma, \Gamma_i^1 \vdash A^* \uparrow A_i^1$ (for all $i \in 1..n_1$) Above

  $\Gamma, \Gamma_i^1; \Delta_{i2}^1, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C_1$ (for all $i \in 1..n_1$) By Lemma 5.59

  $\Gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C_1$ By Proposition 5.55

  $\Gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C_2$ By similar reasoning w.r.t. 2nd premise

	$\mathcal{W}[\mathbf{let}\ \overline{\mathbf{x}} = e'\ \mathbf{in}\ e]$ value	$\wedge$I restricted to values
	$\mathcal{W}[e]$ value	
☞	$\Gamma; \Delta, \overline{\mathbf{x}}{:}A^* \vdash \mathcal{W}[e] \downarrow C_1 \wedge C_2$	By $\wedge$I
☞	$\Gamma; \cdot \vdash y \uparrow A^*$	Above
☞	$\{\Gamma; \Delta\}\ \boxed{\Psi}\ \Gamma; \Delta$	By Ψ-refl

- **Case** $\bot\mathbb{L}$: Show (a).

- **Case** $\vee\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma; \Delta, \overline{\mathbf{y}}{:}D_1 \vdash \mathcal{W}[\mathbf{let}\ \overline{\mathbf{x}} = e'\ \mathbf{in}\ e] \downarrow C \qquad \Gamma; \Delta, \overline{\mathbf{y}}{:}D_2 \vdash \mathcal{W}[\mathbf{let}\ \overline{\mathbf{x}} = e'\ \mathbf{in}\ e] \downarrow C}{\Gamma; \Delta, \overline{\mathbf{y}}{:}D_1 \vee D_2 \vdash \mathcal{W}[\mathbf{let}\ \overline{\mathbf{x}} = e'\ \mathbf{in}\ e] \downarrow C}$

  Use the IH twice.

  - If either use of the IH shows (a), show (a).
  - If both uses show (b), either $\overline{\mathbf{y}} \in FLV(e')$ or $\overline{\mathbf{y}} \in FLV(\mathcal{W}[e])$. The former case motivates the multiplicity of judgments; the latter case can be proved in analogous fashion to the former case. Hence we show only the former case.

IH on first premise	IH on second premise
$\{(\Gamma, \Gamma_i^1; \Delta_{i1}^1, \Delta_{i2}^1) \mid i \in 1..n_1\}\ \boxed{\Psi}\ \Gamma; \Delta, \overline{\mathbf{y}}{:}D_1$	$\{(\Gamma, \Gamma_i^2; \Delta_{i1}^1, \Delta_{i2}^2) \mid i \in i..n_2\}\ \boxed{\Psi}\ \Gamma; \Delta, \overline{\mathbf{y}}{:}D_2$
$\Gamma, \Gamma_i^1; \Delta_{i1}^1 \vdash e' \uparrow A_i^1$ for $i \in 1..n_1$	$\Gamma, \Gamma_i^2; \Delta_{i1}^2 \vdash e' \uparrow A_i^2$ for $i \in 1..n_2$
$\Gamma, \Gamma_i^1; \Delta_{i2}^1, \overline{\mathbf{x}}{:}A_i^1 \vdash \mathcal{W}[e] \downarrow C$ for $i \in 1..n_1$	$\Gamma, \Gamma_i^2; \Delta_{i2}^2, \overline{\mathbf{x}}{:}A_i^2 \vdash \mathcal{W}[e] \downarrow C$ for $i \in 1..n_2$

  Unlike the $\wedge$I case we have no value restriction—e' is not necessarily a variable, so we may not have a single best type for e', and cannot join anything now.

  We start by collecting all our derivations together; this is mere bookkeeping. Let $n = n_1 + n_2$. Think of laying out all objects superscripted 1, then all objects superscripted 2. To be painfully precise, for $i \in 1..n_1$, let $\Delta_{i1} = \Delta_{i1}^1, \overline{\mathbf{y}}{:}D_1$; for $i \in (n_1+1)..(n_1+n_2)$, let $\Delta_{i1} = \Delta_{(i-n_1)1}^2, \overline{\mathbf{y}}{:}D_2$. Doing the same for Γ_i and Δ_{i2}, we get

☞	$\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i$	for $i \in 1..n$
☞	$\Gamma, \Gamma_i; \Delta_{i2}, \overline{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C$	for $i \in 1..n$

  We now need to show $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\}\ \boxed{\Psi}\ \Gamma; \Delta$. We have

$$\{(\Gamma, \Gamma_i^1; \Delta_{i1}^1, \Delta_{i2}^1) \mid i \in 1..n_1\}\ \boxed{\Psi}\ \Gamma; \Delta, \overline{\mathbf{y}}{:}D_1$$
$$\text{and}\quad \{(\Gamma, \Gamma_j^2; \Delta_{j1}^1, \Delta_{j2}^2) \mid j \in (n_1{+}1)..n_2\}\ \boxed{\Psi}\ \Gamma; \Delta, \overline{\mathbf{y}}{:}D_2$$

  That is, the set of contexts coming out of the first premise joins $\Gamma; \Delta, \overline{\mathbf{y}}{:}D_1$, and the set coming out of the second premise joins $\Gamma; \Delta, \overline{\mathbf{y}}{:}D_2$. By rule $\vee\Psi$,

$$\begin{array}{l}\{(\Gamma, \Gamma_i^1; \Delta_{i1}^1, \Delta_{i2}^1) \mid i \in 1..n_1\} \quad \cup \\ \{(\Gamma, \Gamma_j^2; \Delta_{j1}^1, \Delta_{j2}^2) \mid j \in (n_1{+}1)..n_2\}\end{array}\ \boxed{\Psi}\ \Gamma; \Delta, \overline{\mathbf{y}}{:}D_1 \vee D_2$$

Bookkeeping gives us

$$☞ \quad \{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \; \boxed{\Psi} \; \Gamma; \Delta, \bar{\mathbf{y}}{:}D_1 \vee D_2$$

- **Case** $\Sigma\mathbb{L}$: $\boxed{\mathcal{D} :: \dfrac{\Gamma, a{:}\gamma; \Delta', \bar{\mathbf{y}}{:}B \vdash \mathcal{W}[\mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow C}{\Gamma; \Delta', \bar{\mathbf{y}}{:}\Sigma a{:}\gamma.\, B \vdash \mathcal{W}[\mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow C}}$

  Apply the IH. If it shows part (a), for all $\Gamma^*, \Delta^*, e^*, D$ we have $\Gamma, a{:}\gamma, \Gamma^*; \Delta', \bar{\mathbf{y}}{:}B, \Delta^* \vdash e^* \downarrow D$. By $\Sigma\mathbb{L}$, we have $\Gamma, \Gamma^*; \Delta', \bar{\mathbf{y}}{:}\Sigma a{:}\gamma.\, B, \Delta^* \vdash e^* \downarrow D$, which shows (a).

  If the IH shows part (b), we have $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \; \boxed{\Psi} \; \Gamma, a{:}\gamma; \Delta', \bar{\mathbf{y}}{:}B$. By $\Sigma\Psi$, we have

  $$☞ \quad \{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \; \boxed{\Psi} \; \Gamma; \Delta', \bar{\mathbf{y}}{:}\Sigma a{:}\gamma.\, B$$

  The rest of what was to be shown is identical to what was shown by the IH.

- **Cases** $\wedge\mathbb{L}_{1,2}$, $\Pi\mathbb{L}$, $\supset\mathbb{L}$, $\otimes\mathbb{L}$: Similar to the $\Sigma\mathbb{L}$ case. □

Corollary 5.62 (Permutation). *If* $\Gamma; \Delta \vdash \mathcal{W}[\mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, e] \downarrow C$ *then* $\Gamma; \Delta \vdash \mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e] \downarrow C$.

Proof. By Lemma 5.60.
If (a), let $\Gamma^* = \Delta^* = \cdot$ and $e^* = \mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e]$, yielding $\Gamma; \Delta \vdash \mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e]$.
If (b), apply let to each pair of derivations

$$\Gamma, \Gamma_i; \Delta_{i1} \vdash e' \uparrow A_i \qquad \Gamma, \Gamma_i; \Delta_{i2}, \bar{\mathbf{x}}{:}A_i \vdash \mathcal{W}[e] \downarrow C$$

yielding $\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2} \vdash \mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e] \downarrow C$ for $i \in 1..n$. Part (b) also gives us $\{(\Gamma, \Gamma_i; \Delta_{i1}, \Delta_{i2}) \mid i \in 1..n\} \; \boxed{\Psi} \; \Gamma; \Delta$. By Proposition 5.55, $\Gamma; \Delta \vdash \mathbf{let}\, \bar{\mathbf{x}} = e' \,\mathbf{in}\, \mathcal{W}[e] \downarrow C$. □

Corollary 5.63 (Permutation of a Subterm). *If* $e = \mathcal{C}[\mathcal{W}[\mathbf{let}\, \bar{\mathbf{x}} = e_1 \,\mathbf{in}\, e_2]]$ *and either*

(1) $\Gamma; \Delta \vdash e \downarrow C$*, or*

(2) $\Gamma; \Delta \vdash e \uparrow C$ *and* $\mathcal{C} \neq [\,]$

then $\Gamma; \Delta \vdash \mathcal{C}[\mathbf{let}\, \bar{\mathbf{x}} = e_1 \,\mathbf{in}\, \mathcal{W}[e_2]] \downarrow\uparrow C$.

Proof. By induction on the given derivation.
Case (1):

- If $\mathcal{C} \neq [\,]$, apply the induction hypothesis to the appropriate premise with a smaller $\mathcal{C}$; if that premise is synthesizing, the smaller $\mathcal{C}$ cannot be $[\,]$ by reasoning similar to that in the proof of Corollary 5.49; finally, reapply the rule.
- If $\mathcal{C} = [\,]$, Corollary 5.62 suffices.

Case (2): As in case (1) where $\mathcal{C} \neq [\,]$. □

Corollary 5.64 (Permutation of a Subterm, Slack). *If* $e = \mathcal{C}[\mathcal{W}[\mathbf{let} \sim\bar{\mathbf{x}} = v_1 \,\mathbf{in}\, e_2]]$ *and either*

(1) $\Gamma; \Delta \vdash e \downarrow C$*, or*

(2) $\Gamma; \Delta \vdash e \uparrow C$ *and* $\mathcal{C} \neq []$

then $\Gamma; \Delta \vdash \mathcal{C}[\mathbf{let} \sim\overline{\mathbf{x}} = v_1 \mathbf{~in~} \mathcal{W}[e_2]] \downarrow\uparrow C$.

Proof. By induction on the given derivation.
Case (1):

- If $\mathcal{C} \neq []$, apply the induction hypothesis to the appropriate premise with a smaller $\mathcal{C}$; if that premise is synthesizing, the smaller $\mathcal{C}$ cannot be $[]$ by reasoning similar to that in the proof of Corollary 5.49; finally, reapply the rule.
- If $\mathcal{C} = []$, use Lemma 5.54.

Case (2): As in case (1) where $\mathcal{C} \neq []$. □

Value permutation

In this section, we show that given a term $\mathbf{let}\ \overline{\mathbf{y}} = e_1 \mathbf{~in~let}\ \overline{\mathbf{x}} = v_2 \mathbf{~in~} e_3$ where $\overline{\mathbf{y}}$ is not free in v_2 (and certain other conditions hold), swapping the bindings preserves typing. We first prove an inversion lemma. The final result, Lemma 5.69, is shown for all variants of slackness (zero, one, or both bindings slack).

Lemma 5.65 (Value Binding Inversion)**.** *If* $\Gamma; \Delta_3 \vdash \mathbf{let}\ \overline{\mathbf{x}} = v_2 \mathbf{~in~} e_3 \downarrow C$ *where* v_2 *is a synthesizing form that is not some* $(v_2' : As)$ *then there exists* B *such that*

$$\Gamma; \cdot \vdash v_2 \uparrow B \quad \textit{and} \quad \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C$$

Remark 5.66. The requirement that v_2 is a value is key. Otherwise the $\bot\mathbb{L}$ case would fail—moving the binding of a non-value can change the order of computational effects, and the (only) computational effect in the language is nontermination, which $\bot\mathbb{L}$ takes account of.

Proof. By induction on the second derivation.

By the condition that v_2 is not an annotation and the assumption that no binding has just a linear variable on the right hand side, v_2 must be some ordinary variable y, allowing us to apply Lemmas 5.3 and 5.4 at will.

- **Case** let: $\mathcal{D} :: \dfrac{\Gamma; \cdot \vdash v_2 \uparrow B \qquad \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C}{\Gamma; \Delta_3 \vdash \mathbf{let}\ \overline{\mathbf{x}} = v_2 \mathbf{~in~} e_3 \downarrow C}$

  We need not use the IH.

  ☞ $\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C$ Subd.
  ☞ $\Gamma; \cdot \vdash v_2 \uparrow B$ Subd.

- **Case** contra: $\mathcal{D} :: \dfrac{\overline{\Gamma} \models \bot \qquad \Gamma \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3\ \mathsf{ok} \qquad \Delta_3 \Vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3\ \mathsf{ok}}{\Gamma; \Delta_3 \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C}$

	$\Gamma \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3\ \mathsf{ok}$	Subd.
	$\Gamma \vdash v_2\ \mathsf{ok}$	By defn. of $\vdash \ldots\ \mathsf{ok}$
☞	$\exists B.\ \Gamma; \cdot \vdash v_2 \uparrow B$	By Lemma 5.4
	$\Delta_3 \Vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3\ \mathsf{ok}$	Subd.
	$\cdot \Vdash v_2\ \mathsf{ok}$	By defn. of $\Vdash$ (given that $v_2 = y$ for some y)
	$\Delta_3, \overline{\mathbf{x}}{:}B \Vdash e_3\ \mathsf{ok}$	By defn. of $\Vdash$
	$\overline{\Gamma} \models \bot$	Subd.
☞	$\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C$	By contra

- **Case** $\wedge$I: $\mathcal{D} :: \dfrac{\Gamma; \Delta_3 \vdash v \downarrow C_1 \qquad \Gamma; \Delta_3 \vdash v \downarrow C_2}{\Gamma; \Delta_3 \vdash v \downarrow C_1 \wedge C_2}$

  Here $v = (\textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3)$ and e_3 value.

	$\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B_1 \vdash e_3 \downarrow C_1$ and $\Gamma; \cdot \vdash v_2 \uparrow B_1$	By IH
	$\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B_2 \vdash e_3 \downarrow C_2$ and $\Gamma; \cdot \vdash v_2 \uparrow B_2$	By IH
$\exists B.$	$\Gamma; \cdot \vdash v_2 \uparrow B$ and $\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C_1$ and $\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C_2$	By Lemma 5.3
	e_3 value	Given
☞	$\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C_1 \wedge C_2$	By $\wedge$I
☞	$\Gamma; \cdot \vdash v_2 \uparrow B$	Above

- **Case** $\top$I: $\mathcal{D} :: \dfrac{\Gamma \vdash v\ \mathsf{ok} \qquad \Delta_3 \Vdash v\ \mathsf{ok}}{\Gamma; \Delta_3 \vdash v \downarrow \top}$

  Similar to the contra case, with additional (easy) reasoning about values.

- **Case** ΠI: $\mathcal{D} :: \dfrac{\Gamma, a{:}\gamma; \Delta_3 \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C'}{\Gamma; \Delta_3 \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow \Pi a{:}\gamma.\ C'}$

	$\Gamma, a{:}\gamma; \Delta_3 \vdash \textbf{let}\,\overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C'$	Subd.
$\exists B.$	$\Gamma, a{:}\gamma; \cdot \vdash v_2 \uparrow B$ and $\Gamma, a{:}\gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C'$	By IH
	$a \notin dom(\Gamma)$	a fresh
	$\forall x.\ a \notin FV(\Gamma(x))$	Γ cannot refer to index variables not in $dom(\Gamma)$
☞	$\Gamma; \cdot \vdash v_2 \uparrow B$	By var

$$\begin{array}{ll} e_3 \text{ value} & \text{By } (\textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3) \text{ value} \\ ☞\ \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow \Pi a{:}\gamma.\, C' & \text{By } \Pi I \end{array}$$

- **Cases** $\vee I_1$, $\vee I_2$, ΣI, $\otimes I$: Apply the IH and reapply the rule to the checking judgment obtained.
- **Cases** $\wedge\mathbb{L}_1$, $\wedge\mathbb{L}_2$, $\Pi\mathbb{L}$, $\supset\mathbb{L}$: Apply the IH and reapply the rule to the checking judgment obtained.
- **Case** $\vee\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma; \Delta_3, \overline{\mathbf{z}}{:}D_1 \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C \qquad \Gamma; \Delta_3, \overline{\mathbf{z}}{:}D_2 \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C}{\Gamma; \Delta_3, \overline{\mathbf{z}}{:}D_1 \vee D_2 \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C}$

  v_2 is some ordinary variable y so $FLV(v_2) = \{\}$. Thus $\overline{\mathbf{z}} \in FLV(e_3)$.

$$\begin{array}{lll} & \Gamma; \Delta_3, \overline{\mathbf{z}}{:}D_1 \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C & \text{Subd.} \\ \exists B_1. & \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B_1, \overline{\mathbf{z}}{:}D_1 \vdash e_3 \downarrow C \ \text{ and } \ \Gamma; \cdot \vdash y_2 \uparrow B_1 & \text{By IH} \\ & \Gamma; \Delta_3, \overline{\mathbf{z}}{:}D_2 \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C & \text{Subd.} \\ \exists B_2. & \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B_2, \overline{\mathbf{z}}{:}D_2 \vdash e_3 \downarrow C \ \text{ and } \ \Gamma; \cdot \vdash y_2 \uparrow B_2 & \text{By IH} \\ ☞\ \exists B. & \Gamma; \cdot \vdash y_2 \uparrow B \ \text{ and } \ \Gamma \vdash B \uparrow B_1, \ \ \Gamma \vdash B \uparrow B_2 & \text{By Lemma 5.3} \\ & \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B, \overline{\mathbf{z}}{:}D_1 \vdash e_3 \downarrow C & \text{By Props. 5.57, 5.55} \\ & \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B, \overline{\mathbf{z}}{:}D_2 \vdash e_3 \downarrow C & \text{By Props. 5.57, 5.55} \\ ☞ & \Gamma; \Delta_3, \overline{\mathbf{x}}{:}B, \overline{\mathbf{z}}{:}D_1 \vee D_2 \vdash e_3 \downarrow C & \text{By } \vee\mathbb{L} \end{array}$$

- **Case** $\bot\mathbb{L}$: Show $\Delta, \overline{\mathbf{z}}{:}D \Vdash e_3$ ok as in contra, then apply $\bot\mathbb{L}$.
- **Case** $\Sigma\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma, a{:}\gamma; \Delta_3, \overline{\mathbf{z}}{:}D \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C}{\Gamma; \Delta_3, \overline{\mathbf{z}}{:}\Sigma a{:}\gamma.\, D \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C}$

  $\Gamma, a{:}\gamma; \Delta_3, \overline{\mathbf{z}}{:}D \vdash \textbf{let}\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C$ Subd.

  v_2 is some ordinary variable y so $FLV(v_2) = \{\}$. Thus $\overline{\mathbf{z}} \in FLV(e_3)$.

$$\begin{array}{lll} & \Gamma, a{:}\gamma; \Delta_3, \overline{\mathbf{z}}{:}D, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C & \text{By IH} \\ & \quad \text{and } \ \Gamma, a{:}\gamma; \cdot \vdash v_2 \uparrow B & \\ ☞ & \Gamma; \cdot \vdash v_2 \uparrow B & \text{By reasoning analogous to the } \Pi I \text{ case} \\ ☞ & \Gamma; \Delta_3, \overline{\mathbf{z}}{:}\Sigma a{:}\gamma.\, D, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C & \text{By } \Sigma\mathbb{L} \end{array}$$

- **Case** $\otimes\mathbb{L}$: Similar to the $\Sigma\mathbb{L}$ case. □

Lemma 5.67 (Value Permutation, First Binding Ordinary). *If*

$$\Gamma; \Delta \vdash \textbf{let}\ \overline{\mathbf{y}} = e_1\ \textbf{in let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \textbf{in}\ e_3 \downarrow C$$

where $\overline{\mathbf{y}} \notin FV(v_2)$ *and* v_2 *is a synthesizing form*
and, if the binding of $\overline{\mathbf{x}}$ *is ordinary,* v_2 *does not have the form* $(v_2 : As)$*,*
then $\Gamma; \Delta \vdash \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$.

Proof. By induction on the given derivation. At times, we silently partition Δ into $\Delta_1, \Delta_2, \Delta_3$ where $\Delta_1 \Vdash e_1\ \mathsf{ok}$, $\Delta_2 \Vdash v_2\ \mathsf{ok}$, $\Delta_3 \Vdash e_3\ \mathsf{ok}$.

- **Case** let: $\mathcal{D} :: \dfrac{\Gamma; \Delta_1 \vdash e_1 \uparrow A \qquad \Gamma; \Delta_2, \Delta_3, \overline{\mathbf{y}}{:}A \vdash \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C}{\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C}$

  $\Gamma; \Delta_1 \vdash e_1 \uparrow A$ Subd.

  $\Gamma; \Delta_2, \Delta_3, \overline{\mathbf{y}}{:}A \vdash \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C$ Subd.

  Now we split on whether the binding of $\overline{\mathbf{x}}$ is slack.

  - The binding of $\overline{\mathbf{x}}$ is ordinary:

    $\Gamma; \cdot \vdash v_2 \uparrow B$ and $\Gamma; \Delta_3, \overline{\mathbf{x}}{:}B, \overline{\mathbf{y}}{:}A \vdash e_3 \downarrow C$ By Lemma 5.65

    $\Gamma; \Delta_1 \vdash e_1 \uparrow A$ Subd.

    $\Gamma; \Delta_1, \Delta_3, \overline{\mathbf{x}}{:}B \vdash \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$ By let

    $\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \mathbf{let}\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$ By let

  - The binding of $\overline{\mathbf{x}}$ is slack:

    $\Gamma; \Delta_2, \Delta_3, \overline{\mathbf{y}}{:}A, \sim\overline{\mathbf{x}} = v_2 \vdash e_3 \downarrow C$ By Lemma 5.54

    $\Gamma; \Delta_1, \Delta_2, \Delta_3, \sim\overline{\mathbf{x}} = v_2 \vdash \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$ By let

    $\Gamma; \Delta_1, \Delta_2, \Delta_3 \vdash \mathbf{let}\,\sim\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$ By let$\sim$

- **Case** contra: We have in the premise of contra $\Delta \Vdash \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3\ \mathsf{ok}$. Since $\overline{\mathbf{y}} \notin FV(v_2)$, it follows from the definition of $\Vdash$ that $\Delta \Vdash \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3\ \mathsf{ok}$. Applying contra gives the result.

- **Case** $\wedge$I: It follows from

$$\mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3\ \mathsf{value}$$

  that $e_1\ \mathsf{value}$ and $e_3\ \mathsf{value}$. Therefore $(\mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\,\mathbf{in}\,\mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3)\ \mathsf{value}$. Apply the IH to each premise and reapply the rule.

- **Case** $\top$I: By the same reasoning as in the contra case, $\Delta \Vdash \mathbf{let}\,[\sim]\,\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\,\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3\ \mathsf{ok}$. Apply $\top$I; the value restriction is satisfied by the reasoning in the $\wedge$I case above.

- **Case** $\bot\mathbb{L}$: Similar to the $\top$I case.

- **Cases** ΠI, ΣI, $\otimes$I, $\vee$I$_1$, $\vee$I$_2$, $\vee\mathbb{L}$, $\Sigma\mathbb{L}$, $\otimes\mathbb{L}$, $\wedge\mathbb{L}_1$, $\wedge\mathbb{L}_2$, $\Pi\mathbb{L}$, $\supset\mathbb{L}$: IH to each premise, then reapply.

- **Case** sub: Impossible: a let-term can never synthesize, so the first premise could not have been derived. □

Lemma 5.68 (Value Permutation, First Binding Slack). *If* $\Gamma; \Delta \vdash \mathbf{let} \sim\overline{\mathbf{y}} = e_1 \ \mathbf{in}\ \mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C$ *where* $\overline{\mathbf{y}} \notin FV(v_2)$
and, if the binding of $\overline{\mathbf{x}}$ *is ordinary,* v_2 *does not have the form* $(v_2 : As)$,
then $\Gamma; \Delta \vdash \mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$.

Proof. By induction on the given derivation, following the proof of Lemma 5.67 except in the let$\sim$ case.

- **Case** let$\sim$: $\mathcal{D} :: \dfrac{\Gamma; \Delta, \sim\overline{\mathbf{y}} = e_1 \vdash \mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C}{\Gamma; \Delta \vdash \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ \mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C}$

  Now we split on whether the binding of $\overline{\mathbf{x}}$ is slack.

  - The binding of $\overline{\mathbf{x}}$ is ordinary:

$\Gamma; \Delta, \sim\overline{\mathbf{y}} = e_1 \vdash \mathbf{let}\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C$	Subd.
$\Gamma; \cdot \vdash v_2 \uparrow B$ and $\Gamma; \Delta, \sim\overline{\mathbf{y}} = e_1, \overline{\mathbf{x}}{:}B \vdash e_3 \downarrow C$	By Lemma 5.65
$\Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$	By let$\sim$
$\Gamma; \Delta \vdash \mathbf{let}\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$	By let

  - The binding of $\overline{\mathbf{x}}$ is slack:

$\Gamma; \Delta, \sim\overline{\mathbf{y}} = e_1 \vdash \mathbf{let} \sim\overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3 \downarrow C$	Subd.
$\Gamma; \Delta, \sim\overline{\mathbf{y}} = e_1, \sim\overline{\mathbf{x}} = v_2 \vdash e_3 \downarrow C$	By Lemma 5.52
$\Gamma; \Delta, \sim\overline{\mathbf{x}} = v_2 \vdash \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$	By let$\sim$
$\Gamma; \Delta \vdash \mathbf{let} \sim\overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let} \sim\overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3 \downarrow C$	By let$\sim$

    □

Lemma 5.69 (Omnibus Value Permutation). *If* $\Gamma; \Delta \vdash \mathcal{C}[e'] \ \underline{\downarrow\uparrow}\ C$ *where* $e' = \mathbf{let}\ [\sim]\ \overline{\mathbf{y}} = e_1\ \mathbf{in}$ $(\mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ e_3)$ *and* $\overline{\mathbf{y}} \notin FV(v_2)$
and, if the binding of $\overline{\mathbf{x}}$ *is ordinary,* v_2 *does not have the form* $(v_2 : As)$,
then

$$\Gamma; \Delta \vdash \mathcal{C}[\mathbf{let}\ [\sim]\ \overline{\mathbf{x}} = v_2\ \mathbf{in}\ \mathbf{let}\ [\sim]\ \overline{\mathbf{y}} = e_1\ \mathbf{in}\ e_3] \ \underline{\downarrow\uparrow}\ C$$

(This includes all 4 possibilities: neither binding slack, first binding slack, second binding slack, and both bindings slack.)

Proof. By a straightforward induction on the derivation of $\Gamma; \Delta \vdash e \ \underline{\downarrow\uparrow}\ C$.

If $\mathcal{C} = []$ then $\mathcal{C}[e'] = e'$ is a **let**, which cannot synthesize anything, so we have a checking judgment $\Gamma; \Delta \vdash e \downarrow C$. In this case we use Lemma 5.67 (if the first binding is ordinary) or Lemma 5.68 (if the first binding is slack).

Otherwise, we just apply the IH to the appropriate subterm and reapply the rule. For short-circuit rules we use the side condition that $\overline{\mathbf{y}} \notin FV(v_2)$ and the ok-premises of the rule to show that the appropriate ok-premises hold, enabling us to apply the same rule to the new term. □

5.7.5 Results

Recall that $\downarrow^{\mathsf{let}}\uparrow^{\mathsf{let}}$ appear in judgments in the type system for terms with **let**s and other syntactic markers, while $\downarrow^{\mathbb{L}}\uparrow^{\mathbb{L}}$ appear in judgments for terms in the left tridirectional type system. In most of this chapter, $\downarrow$ and $\uparrow$ are used instead of $\downarrow^{\mathsf{let}}$ and $\uparrow^{\mathsf{let}}$.

Definition 5.70. Variables $\overline{\mathbf{x}}$, $\overline{\mathbf{y}}$ in a term e are *disordered* iff $\overline{\mathbf{x}}$ is bound before $\overline{\mathbf{y}}$ but $\overline{\mathbf{y}}$ precedes $\overline{\mathbf{x}}$ (see Def. 5.35).

Lemma 5.71. *If* $\Gamma; \Delta \vdash e \downarrow^{\mathsf{let}}\uparrow^{\mathsf{let}} C$ *then every linear variable in* $\mathrm{dom}(\Delta)$ *appears exactly once in* e*. Moreover, the same holds for every linear variable* bound *in* e*.*

Proof. By induction on the given derivation. □

Lemma 5.72. *If*

brittle: see Def. 5.44

(1) for every subterm e' *of* e*,*

e' *is synthesizing and not a linear variable and not brittle* *iff* e' *is the right hand side of a* **let**

(2) if $\overline{\mathbf{x}}$ *and* $\overline{\mathbf{y}}$ *are used at the same level (that is, there are no intervening bindings) and the use of* $\overline{\mathbf{x}}$ *precedes the use of* $\overline{\mathbf{y}}$*, then the binding of* $\overline{\mathbf{x}}$ *precedes the binding of* $\overline{\mathbf{y}}$

(3) for every subterm e' *of* e*,* $\hookleftarrow(e')$ *has the form* $(v : As)$ *iff it is the right hand side of a* **let**$\sim$

(4) e *contains no prickly subterms (i.e. for all* $\mathcal{C}$*, if* $e = \mathcal{C}[e']$ *then there exists no* $\mathcal{W} \neq []$ *such that* $e = \mathcal{C}'[\mathcal{W}[e']]$*)*

(5) $\Delta \Vdash e\ \mathsf{ok}$

then there exists e^{-1} *such that* $e^{-1} \hookrightarrow L_m + e_m$ *and* $e = L_m$ **in** e_m*. (By Proposition 5.38,* e^{-1} *is the unique such expression.)*

Moreover, given matches ms *instead of some* e*, where conditions (1)–(5) hold for every case arm in* ms *(in (5), with* $\Delta = \cdot$*), there exists* ms^{-1} *such that* $ms^{-1} \hookrightarrow ms$*.*

Proof. By induction on the size of e.

Decompose e into L **in** e^* where L is maximal. We distinguish cases of e^*. Several synthesizing form cases are impossible by (1) (but see the subcases of $e^* = \overline{\mathbf{x}}$): $e^* = x$, $\mathbf{fst}(e_0)$, $\mathbf{snd}(e_0)$, $e_1\, e_2$, u.

- $\boxed{e^* = ()}$ By (5), any bound variable must be used in e^*, but $FLV(e^*) = \{\}$, so $L = \cdot$. Let $e^{-1} = ()$.
- $\boxed{e^* = (e_1, e_2)}$
  - If e_1 antivalue:

    By the syntactic restriction that a **let**-bound variable must appear in a viable position in its scope, every variable bound in L appears in a viable position in (e_1, e_2). The only possible positions—given that e_1 antivalue—are in e_1. Therefore we have $BLV(L) = FLV(e_1)$. By linearity (5), $FLV(e_1) \cap FLV(e_2) = \{\}$.

By IH on L **in** e_1, there exists ${e_1}^{-1} \hookrightarrow L' + e_1'$ where L' **in** $e_1' = L$ **in** e_1. By Proposition 5.16, L' **in** e_1' is maximal. Also maximal is L **in** e_1, since (4) forbids any **let** on e_1. Therefore $L' = L$ and $e_1' = e_1$: we have ${e_1}^{-1} \hookrightarrow L + e_1$.

By IH on e_2, there exists ${e_2}^{-1} \hookrightarrow L_2' + e_2'$ where L_2' **in** $e_2' = e_2$.

It is clear from the definition of $\hookrightarrow$ that ${e_1}^{-1}$ antivalue if and only if e_1 antivalue. Therefore ${e_1}^{-1}$ antivalue. By the appropriate rule, we have $({e_1}^{-1}, {e_2}^{-1}) \hookrightarrow L + (e_1, e_2)$. Let $e^{-1} = ({e_1}^{-1}, {e_2}^{-1})$.

- Otherwise, e_1 prevalue.

  By (2) and (5), L can be decomposed into L_1, L_2 such that $BLV(L_1) \subseteq FLV(e_1)$ and $BLV(L_2) \subseteq FLV(e_2)$. By IH, there exist ${e_1}^{-1}$, ${e_2}^{-1}$ such that ${e_k}^{-1} \hookrightarrow L_k' + e_k'$. Analogous to the e_1 antivalue subcase (reasoning about maximality), $L_k' = L_k$ and $e_k' = e_k$.

  Dual to the previous case, we obtain ${e_1}^{-1}$ prevalue. We have $({e_1}^{-1}, {e_2}^{-1}) \hookrightarrow L_1, L_2 + (e_1, e_2)$ by the appropriate rule. Let $e^{-1} = ({e_1}^{-1}, {e_2}^{-1})$.

- $\boxed{e^* = \lambda x.\, e_0}$ A term of the form $\lambda x.\, e_0$ has no viable subterms. By (5), $L = \cdot$.

  By IH, there exists ${e_0}^{-1}$ such that ${e_0}^{-1} \hookrightarrow e_0$. Let $e^{-1} = \lambda x.\, {e_0}^{-1}$.

- $\boxed{e^* = \textbf{fix}\ u.\, e_0}$ By (5), $L' = \cdot$. By IH, there exists ${e_0}^{-1} \hookrightarrow e_0$. Let $e^{-1} = \textbf{fix}\ u.\, {e_0}^{-1}$.

- $\boxed{e^* = \textbf{case}\ e_0\ \textbf{of}\ ms}$ By (5), $BLV(L') = FLV(e_0)$. By IH, there exists ${e_0}^{-1} \hookrightarrow L' + e_0$. Also by IH, there exists ms^{-1} such that $ms^{-1} \hookrightarrow ms$. Let $e^{-1} = \textbf{case}\ {e_0}^{-1}\ \textbf{of}\ ms^{-1}$.

- $\boxed{e^* = c(e_0)}$ Similar to the $e^* = \textbf{case}\ e_0\ \textbf{of}\ ms$ case, without ms.

- $\boxed{e^* = \bar{\mathbf{x}}}$ Using (5), we have $BLV(L) = FLV(e^*) = \{\bar{\mathbf{x}}\}$. Therefore $L = L', [\sim]\bar{\mathbf{x}} = e'$ for some e' such that $BLV(L') = FLV(e')$. By (1), e' is a synthesizing form. Now we distinguish cases of e':

  - $\boxed{e' = x}$ By (5), $BLV(L') = FLV(e')$, which is empty. Thus $L' = \cdot$. Let $e^{-1} = x$.
  - $\boxed{e' = u}$ Similar to the preceding case.
  - $\boxed{e' = \textbf{fst}(e_0)}$ Apply IH to L' **in** e_0, yielding some ${e_0}^{-1}$. Let $e^{-1} = \textbf{fst}({e_0}^{-1})$.
  - $\boxed{e' = \textbf{snd}(e_0)}$ Similar to the preceding case.
  - $\boxed{e' = e_1\ e_2}$ Broadly similar to the (e_1, e_2) case: distinguish e_1 prevalue and e_1 antivalue subcases; apply the IH and reason about maximality; apply the appropriate rule. The binding of $\bar{\mathbf{x}}$, not present in the (e_1, e_2) case, makes no real difference.
  - $\boxed{e' = (e_0 : As)}$ Similar to the $e' = \textbf{fst}(e_0)$ case.

The cases for ms are straightforward. □

Definition 5.73. A pair of transposed bindings $\bar{\mathbf{x}} = e_1, \ldots, \bar{\mathbf{y}} = e_2$ is *redeemable* if e_2 value.

Proposition 5.74. *If* $e = \mathcal{E}_1[e_1] = \mathcal{E}_2[\overline{\mathbf{y}}]$ *and the hole in* $\mathcal{E}_1$ *precedes the hole in* $\mathcal{E}_2$ *(i.e.* e_1 *precedes* $\overline{\mathbf{y}}$*) then* e_1 value*.*

Proof. By simultaneous induction on $\mathcal{E}_1$ and $\mathcal{E}_2$. □

Lemma 5.75. *Given* L **in** e *with* $L = L_1, \overline{\mathbf{x}} = e_1, L_2, \overline{\mathbf{y}} = e_3, L_3$ *and* $\overline{\mathbf{x}}$*,* $\overline{\mathbf{y}}$ *transposed, there exist adjacent transposed bindings, i.e.* $\overline{\mathbf{x}} = e_1, L_2, \overline{\mathbf{y}} = e_3 = \ldots, \overline{\mathbf{z}}_1 = e_{21}, \overline{\mathbf{z}}_2 = e_{22}, \ldots$ *where* $\overline{\mathbf{z}}_1$ *and* $\overline{\mathbf{z}}_2$ *are transposed.*

Proof. By induction on the length of L_2.

- If $L_2 = \cdot$ then $\overline{\mathbf{x}} = e_1$ and $\overline{\mathbf{y}} = e_3$ are themselves adjacent.
- Otherwise $L_2 = (\overline{\mathbf{x}}_2 = e_2), L_2'$.
  - If $\overline{\mathbf{x}}_2$ precedes $\overline{\mathbf{x}}$ in the unwinding, we have our adjacent transposed bindings.
  - Otherwise $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{x}}_2$. It is given that $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ are transposed, so $\overline{\mathbf{y}}$ precedes $\overline{\mathbf{x}}$. Transitively, $\overline{\mathbf{y}}$ precedes $\overline{\mathbf{x}}_2$, but $\overline{\mathbf{y}}$ is bound after $\overline{\mathbf{x}}_2$ is. Therefore $\overline{\mathbf{x}}_2$ and $\overline{\mathbf{y}}$ are transposed. The length of L_2' is one less than the length of L_2, so we can apply the IH, obtaining the result. □

Lemma 5.76. *If* e *is let-respecting (Def. 5.23) then all adjacent transpositions are redeemable.*

Proof. By induction on e. The interesting case is when $e = \textbf{let}\,\overline{\mathbf{y}} = e_{\overline{\mathbf{y}}}\,\textbf{in let}\,\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}\,\textbf{in}\,e'$ and $\overline{\mathbf{y}}$, $\overline{\mathbf{x}}$ are transposed. Since e is let-respecting $\hookleftarrow(e) = \mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$ and $\hookleftarrow(\textbf{let}\,\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}\,\textbf{in}\,e) = \mathcal{E}_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}]$; by definition of $\hookleftarrow$ and linearity, $\hookleftarrow(\textbf{let}\,\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}\,\textbf{in}\,e) = [\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\hookleftarrow(e) = [\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\,\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}] = \mathcal{E}_{\overline{\mathbf{x}}}[\hookleftarrow(e_{\overline{\mathbf{x}}})]$. We therefore have $\mathcal{E}_{\overline{\mathbf{x}}}[\hookleftarrow(e_{\overline{\mathbf{x}}})] = \mathcal{E}_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}]$ in which $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{y}}$. By Proposition 5.74, $\hookleftarrow(e_{\overline{\mathbf{x}}})$ value. By Propositions 5.8 and 5.6, $e_{\overline{\mathbf{x}}}$ value. Therefore the binding is redeemable. □

Definition 5.77 (Measure for Induction)**.** Let the *measure* of a term e' be defined as

$$\mu(e') = \langle \mathsf{unbound}_{\uparrow}(e'), \mathsf{brittle}(e'), \mathsf{prickly}(e'), \mathsf{transposed}(e') \rangle$$

where

- $\mathsf{unbound}_{\uparrow}(e')$ is the number of subterms of e' in synthesizing form that are not let-bound,
- $\mathsf{brittle}(e')$ is the number of brittle subterms of e', and
- $\mathsf{prickly}(e')$ is the number of prickly subterms of e', and
- $\mathsf{transposed}(e')$ is the number of transposed variable pairs (Definition 5.46) in e',

Order the quadruples lexicographically.

Lemma 5.78 (Typing of Canonical Let-Normal Terms)**.** *If* $\Gamma; \Delta \vdash e' \downarrow^{\mathsf{let}} C$ *where* e' *is let-respecting, then* $\Gamma; \Delta \vdash L$ **in** $e^* \downarrow^{\mathsf{let}} C$ *where* $\hookleftarrow(e') \hookrightarrow L + e^*$ *and* L **in** e^* *is let-respecting.*

This lemma says that, given a term in the let-system, the corresponding canonical term is well typed. Note that Δ may be nonempty and the given term e' may contain free linear variables, if the lemma is invoked on a subterm:

$$\textbf{let}\,\overline{\mathbf{x}} = \mathsf{f}\ \textbf{in}\ \underbrace{\ldots\overline{\mathbf{x}}\ldots}_{e'}$$

In such a situation the unwinding $\hookleftarrow(e')$ contains free linear variables and so does not correspond to any source program, though it is still a direct style term according to Definition 5.7.

Proof. By induction on e' with measure $\mu(e')$ (in Definition 5.77):

$$\mu(e') = \langle \mathsf{unbound}_{\uparrow}(e'), \mathsf{brittle}(e'), \mathsf{prickly}(e'), \mathsf{transposed}(e')\rangle$$

- If there exists any subterm e_0 of e', where e_0 is in synthesizing form and is not the right hand side of a **let** binding, then:

  - Use Corollary 5.49 to obtain a new term e'' such that

    $$\Gamma; \Delta \vdash e'' \downarrow C$$

    It is clear that the first component of $\mu(e'')$ is $\mathsf{unbound}_{\uparrow} - 1$. Therefore $\mu(e'')$ is smaller than $\mu(e')$.

    Corollary 5.49 replaces $\mathcal{E}''[e_1]$ with $\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ \mathcal{E}''[\overline{\mathbf{x}}]$ for some $\mathcal{E}''$.

    The body of the new binding has the form $\mathcal{E}''[\overline{\mathbf{x}}]$; by applying Proposition 5.24 we can show that $\overline{\mathbf{x}}$ appears in evaluation position in $\hookleftarrow(\mathcal{E}''[\overline{\mathbf{x}}])$. To apply the proposition, we show that $\mathcal{E}''[\overline{\mathbf{x}}]$ is let-respecting: We know that $\mathcal{E}''[e_1]$ is let-respecting; by Lemma 5.29, $\mathcal{E}''[\overline{\mathbf{x}}]$ is let-respecting (since the resulting term is well typed, $\mathit{FLV}(e_1) \subseteq \mathit{FLV}(\mathcal{E}''[e_1])$).

    Next, we show that outer bindings remain let-respecting: By Proposition 5.10,

    $$\hookleftarrow(\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ \mathcal{E}''[\overline{\mathbf{x}}]) = \hookleftarrow(\mathcal{E}''[e_1])$$

    By Proposition 5.26, the unwinding of $\mathcal{C}[\textbf{let}\,\overline{\mathbf{x}} = e_1\ \textbf{in}\ \mathcal{E}''[\overline{\mathbf{x}}]]$ is identical to the unwinding of $\mathcal{C}[\mathcal{E}''[e_1]]$ for any $\mathcal{C}$. Thus, the let-respecting property of all outer bindings is preserved.

    The result follows by IH on e''.

- Otherwise, if there exists any brittle subterm, i.e. $e' = \textbf{let}\,\overline{\mathbf{x}} = (v_1 : As)\ \textbf{in}\ e_2$ where $e = \mathcal{C}[e']$:

  - Use Corollary 5.51 to show that $\mathcal{C}[\textbf{let}\ {\sim}\overline{\mathbf{x}} = (v_1 : As)\ \textbf{in}\ e_2]$ is well typed. This merely converts an ordinary binding to a slack binding, so it reduces $\mathsf{brittle}$ while leaving $\mathsf{unbound}_{\uparrow}$ unchanged. Clearly, simply making a binding slack does not affect whether a term is let-respecting. The result follows by IH.

- Otherwise, if there exists any prickly subterm (e.g. $e_1(\textbf{let}\,\overline{\mathbf{x}} = e_2\ \textbf{in}\ e_3)$ where e_1 $\mathsf{prevalue}$):

- Permute the violating **let**/**let**∼ outward: replace $\mathcal{W}[\textbf{let}\,\overline{\mathbf{x}} = e_2 \textbf{ in } e_3]$ with $\textbf{let}\,\overline{\mathbf{x}} = e_2 \textbf{ in } \mathcal{W}[e_3]$. By Corollary 5.63 or Corollary 5.64,

$$\Gamma; \Delta \vdash e'' \downarrow C$$

  The number of unbound synthesizing subterms and the number of brittle subterms remain zero, and the number of prickly subterms is reduced by 1. Therefore $\mu(e'')$ is smaller than $\mu(e')$.

  To show that e'' is let-respecting, we first show that the lifted binding is let-respecting:

  * It must be the case that $\mathcal{W}$ is an evaluation context. To see why, suppose $\mathcal{W} = \check{e}\mathcal{W}'$. Since $\mathsf{unbound}_\uparrow = 0$, the $\check{e}$ cannot be a synthesizing form other than a linear variable, which is a value; supposing $\mathcal{W}'$ is an evaluation context, $v\mathcal{W}'$ is an evaluation context.
    The original term is let-respecting, so its subterm $\mathcal{W}[\textbf{let}\,\overline{\mathbf{x}} = e_2 \textbf{ in } e_3]$ is too. By Lemma 5.33, $\mathcal{W}[e_3]$ is let-respecting.
    By Proposition 5.24, there exists $\mathcal{E}'$ such that $\hookleftarrow(\mathcal{W}[e_3]) = \mathcal{E}'[\hookleftarrow(e_3)]$. Since the original term was let-respecting, there exists $\mathcal{E}_3$ such that $\hookleftarrow(e_3) = \mathcal{E}_3[\overline{\mathbf{x}}]$. Thus $\hookleftarrow(\mathcal{W}[e_3]) = \mathcal{E}'[\mathcal{E}_3[\overline{\mathbf{x}}]] = (\mathcal{E}[\mathcal{E}_3])[\overline{\mathbf{x}}]$: the binding $\textbf{let}\,\overline{\mathbf{x}} = e_2 \textbf{ in } \mathcal{W}[e_3]$ is let-respecting.
  * Showing that all other bindings are let-respecting is analogous to the letification phase above, using Proposition 5.10, etc.

  The result follows by IH on e''.

- Otherwise, if there exists any subterm $e_0 = L \textbf{ in } e$ such that any bindings in L are transposed, by Lemma 5.75 there exist *adjacent* transposed bindings:

$$L = L_1, ([\sim]\overline{\mathbf{y}} = e_{\overline{\mathbf{y}}}), ([\sim]\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}), L_2$$

  where either or both of the bindings of $\overline{\mathbf{y}}$ and $\overline{\mathbf{x}}$ are slack, and $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{y}}$ in $\hookleftarrow(L_2 \textbf{ in } e)$.

  - By Lemma 5.76, $e_{\overline{\mathbf{x}}}$ value.
    Exchange the binding of $\overline{\mathbf{x}}$ with the binding of $\overline{\mathbf{y}}$, yielding

$$e_0' = (L_1, ([\sim]\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}), ([\sim]\overline{\mathbf{y}} = e_{\overline{\mathbf{y}}}), L_2) \textbf{ in } e$$

    We know that $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{y}}$ in $\hookleftarrow(L_2 \textbf{ in } e)$.
    We need to show that $\overline{\mathbf{y}} \notin FV(e_{\overline{\mathbf{x}}})$. Suppose it *were* free in $e_{\overline{\mathbf{x}}}$. Assuming $\overline{\mathbf{y}}$ appears linearly, $\overline{\mathbf{y}}$ cannot also appear free in $L_2 \textbf{ in } e$. By Proposition 5.21, $\overline{\mathbf{y}}$ is not free in $\hookleftarrow(L_2 \textbf{ in } e)$. But we know that $\overline{\mathbf{x}}$ precedes $\overline{\mathbf{y}}$ in $\hookleftarrow(L_2 \textbf{ in } e)$, so both $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ must be free in $\hookleftarrow(L_2 \textbf{ in } e)$, a contradiction.
    Assuming $\overline{\mathbf{y}}$ appears linearly, $\overline{\mathbf{y}} \notin FV(e_{\overline{\mathbf{x}}})$. By Lemma 5.69, $\Gamma; \Delta \vdash \mathcal{C}[e_0'] \downarrow C$. ($e'$ has no brittle subterms, so the condition about v_2 is satisfied.)
    Here, we only exchange two **let**s, so the numbers of unbound synthesizing subterms, brittle subterms, and prickly subterms do not change—they are still 0. Moreover, transposition is defined in terms of the unwinding, so swapping two transposed bindings does not affect whether any other variable pair is transposed. Thus

$$\mu(\mathcal{C}[e_0']) = \langle 0, 0, 0, \mathsf{transposed}(e') - 1 \rangle$$

which is smaller than $\mu(e')$.

We now show that e' is let-respecting. To show that outer bindings are still let-respecting, the same argument used in the Letify and Permute steps suffices. Now we show that the exchanged bindings are let-respecting:

* *Show* $\hookleftarrow(\mathbf{let}\,\overline{\mathbf{y}} = e_{\overline{\mathbf{y}}}\,\mathbf{in}\,e) = \mathcal{E}'_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$.

  $$\begin{aligned}\mathbf{let}\,\overline{\mathbf{y}} = e_{\overline{\mathbf{y}}}\,\mathbf{in}\,e &= [\hookleftarrow(e_{\overline{\mathbf{y}}})/\overline{\mathbf{y}}]\hookleftarrow(e) && \text{By definition of } \hookleftarrow\\ &= [\hookleftarrow(e_{\overline{\mathbf{y}}})/\overline{\mathbf{y}}]\,\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}] && \text{Original } \overline{\mathbf{x}}\text{-binding is let-respecting}\end{aligned}$$

  Since $\overline{\mathbf{y}}$ follows $\overline{\mathbf{x}}$ in $\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$, substituting a term for $\overline{\mathbf{y}}$ will preserve $\overline{\mathbf{x}}$'s evaluation position: there exists $\mathcal{E}'_{\overline{\mathbf{x}}}$ such that $[\hookleftarrow(e_{\overline{\mathbf{y}}})/\overline{\mathbf{y}}]\,\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}] = \mathcal{E}'_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$.

* *Show* $\hookleftarrow(e) = \mathcal{E}'_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}]$.

  The original $\overline{\mathbf{y}}$-binding is let-respecting, so $\hookleftarrow(\mathbf{let}\,\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}\,\mathbf{in}\,e) = \mathcal{E}_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}]$. By definition of $\hookleftarrow$, it is also equal to $[\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\hookleftarrow(e)$; the original

  $$\begin{aligned}\mathcal{E}_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}] &= \hookleftarrow(\mathbf{let}\,\overline{\mathbf{x}} = e_{\overline{\mathbf{x}}}\,\mathbf{in}\,e) && \text{Original } \overline{\mathbf{y}}\text{-binding is let-respecting}\\ &= [\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\hookleftarrow(e) && \text{By definition of } \hookleftarrow\\ &= [\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\,\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}] && \text{Original } \overline{\mathbf{x}}\text{-binding is let-respecting}\end{aligned}$$

  The only difference between $[\hookleftarrow(e_{\overline{\mathbf{x}}})/\overline{\mathbf{x}}]\,\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$, in which $\overline{\mathbf{y}}$ is in evaluation position, and $\mathcal{E}_{\overline{\mathbf{x}}}[\overline{\mathbf{x}}]$ is that a value ($e_{\overline{\mathbf{x}}}$) is replaced by $\overline{\mathbf{x}}$. This preserves all evaluation positions, including $\overline{\mathbf{y}}$'s. Therefore $\hookleftarrow(e) = \mathcal{E}'_{\overline{\mathbf{y}}}[\overline{\mathbf{y}}]$ for some $\mathcal{E}'_{\overline{\mathbf{y}}}$.

The result follows by IH on $\mathcal{C}[e'_0]$.

- Otherwise, we have $\mu(e') = \langle 0, 0, 0, 0\rangle$. By Lemma 5.72 (using Lemma 5.71 to satisfy condition (5) of Lemma 5.72), e' is precisely the canonical let-translation $L\ \mathbf{in}\ e^*$. It is given that $\Gamma; \Delta \vdash e' \downarrow C$, so $\Gamma; \Delta \vdash L\ \mathbf{in}\ e^* \downarrow C$. □

The next lemma takes us from a left tridirectional typing derivation of e to a let-system typing derivation of a term $e' \equiv_{\mathsf{let}} e$, where e' is not guaranteed to be canonical: for example, if $e = x$ and the given left tridirectional derivation of $\ldots \vdash e \downarrow^{\mathbb{L}} C$ just applies var and sub, e' will also be x. Alternatively, given a derivation that does apply direct$\mathbb{L}$, e' will be $\mathbf{let}\,\overline{\mathbf{x}} = x\,\mathbf{in}\,\overline{\mathbf{x}}$.[9]

Lemma 5.79 (Let-System Completeness). *If* $\Gamma; \Delta \vdash e\ \underline{\downarrow^{\mathbb{L}}\uparrow^{\mathbb{L}}}\ C$ *and* $\Delta \Vdash e$ ev/ok *then* $\Gamma; \Delta \vdash e'\ \underline{\downarrow^{\mathsf{let}}\uparrow^{\mathsf{let}}}\ C$ *where*
(i) $e' \equiv_{\mathsf{let}} e$,
(ii) e' *is let-respecting.*

$\Delta \Vdash e$ ev/ok if $\Delta \Vdash e$ ok and each $\overline{\mathbf{x}}$ in Δ is in evaluation position in e (Def. 3.20).

Proof. By induction on the derivation of $\Gamma; \Delta \vdash e\ \underline{\downarrow^{\mathbb{L}}\uparrow^{\mathbb{L}}}\ C$. The condition $\Delta \Vdash e$ is always easy to show when applying the IH, so we elide it.

Parts (i) and (ii) are straightforward syntactic properties:

- Since $\equiv_{\mathsf{let}}$ is congruent, (i) is almost always easy to show: for example, in $\to$E we have $e = e_1 e_2$; by IH on each subderivation, $e'_1 \equiv_{\mathsf{let}} e_1$ and $e'_2 \equiv_{\mathsf{let}} e_2$, from which $e' = e'_1 e'_2 \equiv_{\mathsf{let}} e_2$.

- Likewise, the reasoning to show (ii) is compositional except where a **let** is introduced.

[9] Given a synthesis derivation $\ldots \vdash x \uparrow^{\mathbb{L}} C$, the lemma will simply yield $\ldots \vdash x \uparrow^{\mathsf{let}} C$.

The above reasoning suffices to show (i)–(ii) in almost all cases. The exception is direct$\mathbb{L}$, where we add a **let** to e to get e', and must show that e' is let-equivalent and that the new binding is let-respecting.

In most cases, we use the following strategy to obtain $\Gamma; \Delta \vdash e' \underline{\downarrow^{\mathsf{let}}\uparrow^{\mathsf{let}}} C$:

1. Apply the IH to each subderivation, yielding a new term typed in the let-normal system.

2. Apply the let-normal system rule corresponding to the left tridirectional system rule. (Recall that the two systems are almost identical.)

For example, for var (left tridirectional system) we just apply var (let-normal system). We are not required to reach a canonical let-normal term in this lemma; let Theorem 5.80 take care of that. Our job at the moment is to take the left tridirectional typing derivation and get a let-normal typing derivation for a (probably) **let**-strewn term.

There are two classes of exceptions where we cannot follow the easy strategy. The first comprises direct$\mathbb{L}$, which is of course not present in the let-normal system; there we must add a **let** and apply rule let. The second comprises the subject-duplicating rules $\wedge$I, $\vee\mathbb{L}$, and $\wedge$-ct. There we apply the IH twice to the *same* term, yielding two possibly different terms: an e_1' from the IH on the first subderivation and an e_2' from the second. We have $e' \equiv_{\mathsf{let}} e_1'$ and $e' \equiv_{\mathsf{let}} e_2'$, and therefore $e_1' \equiv_{\mathsf{let}} e_2'$, but if direct$\mathbb{L}$ was not applied identically in each subderivation, e_1' and e_2' will have **let**s in different places. Fortunately, Lemma 5.78 gives us a way to unify e_1' and e_2'—into their canonical let-normal form.[10] (The fact that Lemma 5.78 only works on checking judgments is no barrier, since all the subject-duplicating rules are checking rules.)

- **Case** direct$\mathbb{L}$: $\mathcal{D} :: \dfrac{\Gamma; \Delta_1 \vdash e_1 \uparrow A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathcal{E}[\overline{\mathbf{x}}] \downarrow C}{\Gamma; \Delta_1, \Delta_2 \vdash \mathcal{E}[e_1] \downarrow C}$

  IH on $\mathcal{E}[\overline{\mathbf{x}}]$ yields some e_2'.

  (i):

$e_1' \equiv_{\mathsf{let}} e_1$	By IH (i)
$e_2' \equiv_{\mathsf{let}} \mathcal{E}[\overline{\mathbf{x}}]$	By IH (i)
$\mathbf{let}\ \overline{\mathbf{x}} = e_1'\ \mathbf{in}\ e_2' \equiv_{\mathsf{let}} \mathbf{let}\ \overline{\mathbf{x}} = e_1\ \mathbf{in}\ \mathcal{E}[\overline{\mathbf{x}}]$	$\equiv_{\mathsf{let}}$ congruent
$\equiv_{\mathsf{let}} [e_1/\overline{\mathbf{x}}]\ \mathcal{E}[\overline{\mathbf{x}}]$	By defn. of $\equiv_{\mathsf{let}}$
$\Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash \mathcal{E}[\overline{\mathbf{x}}] \downarrow^{\mathbb{L}} C$	Subd.
$\Delta_2, \overline{\mathbf{x}}{:}A \Vdash \mathcal{E}[\overline{\mathbf{x}}]$ ok	By Proposition 3.19
$[e_1/\overline{\mathbf{x}}]\ \mathcal{E}[\overline{\mathbf{x}}] = \mathcal{E}[e_1]$	By substitution
(i)☞ $\mathbf{let}\ \overline{\mathbf{x}} = e_1'\ \mathbf{in}\ e_2' \equiv_{\mathsf{let}} \mathcal{E}[e_1]$	Transitivity and substitution

  (ii): By IH, e_1 and e_2' are let-respecting. However, to show that $\mathbf{let}\ \overline{\mathbf{x}} = e_1'\ \mathbf{in}\ e_2'$ is let-respecting we also need to show that $\hookleftarrow(e_2) = \mathcal{E}'[\overline{\mathbf{x}}]$ for some $\mathcal{E}'$. By IH, we have $e_2' \equiv_{\mathsf{let}} \mathcal{E}[\overline{\mathbf{x}}]$. By Proposition 5.10, $\hookleftarrow(e_2') = \hookleftarrow(\mathcal{E}[\overline{\mathbf{x}}])$, which by Proposition 5.9 equals $\mathcal{E}[\overline{\mathbf{x}}]$. Thus $\hookleftarrow(e_2') = \mathcal{E}[\overline{\mathbf{x}}]$: the new binding is let-respecting.

[10] This is why we had to define $\overline{\mathbf{x}} \hookrightarrow \ldots$: we need to translate subterms with free linear variables.

- **Case** $\wedge$I: $\mathcal{D} :: \dfrac{\Gamma;\Delta \vdash v \downarrow C_1 \qquad \Gamma;\Delta \vdash v \downarrow C_2}{\Gamma;\Delta \vdash v \downarrow C_1 \wedge C_2}$

  By IH on $\Gamma;\Delta \vdash v \downarrow C_1$, we have $\Gamma;\Delta \vdash v'_1 \downarrow C_1$, and likewise $\Gamma;\Delta \vdash v'_2 \downarrow C_2$.

  (ii) satisfies the condition of Lemma 5.78, which we now apply to v'_1 yielding $\Gamma;\Delta \vdash v''_1 \downarrow C_1$, where $\hookleftarrow(v'_1) \hookrightarrow v''_1$, and to v'_2 yielding $\Gamma;\Delta \vdash v''_2 \downarrow C_2$ where $\hookleftarrow(v'_2) \hookrightarrow v''_2$.

  We need to show that v''_1 and v''_2 are identical.

$v'_1 \equiv_{\mathsf{let}} v$	By IH
$v'_2 \equiv_{\mathsf{let}} v$	By IH
$v'_1 \equiv_{\mathsf{let}} v'_2$	By symmetry and transitivity of $\equiv_{\mathsf{let}}$
$\hookleftarrow(v'_1) \equiv_{\mathsf{let}} \hookleftarrow(v'_2)$	By Proposition 5.10
$v''_1 = v''_2$	By Proposition 5.36

  Let $v' = v''_1 = v''_2$. By Proposition 5.18, $v \equiv_{\mathsf{let}} v'$.

  By Proposition 5.6, v' value. By $\wedge$I, $\Gamma;\Delta \vdash v' \downarrow C_1 \wedge C_2$.

  Condition (i) follows from $v \equiv_{\mathsf{let}} v'$ above. Condition (ii) follows from Lemma 5.78.

- **Case** $\vee\mathbb{L}$: Similar to the previous case (without the reasoning about terms being values).

- **Case** contra: Let $e' = e$. By contra, $\Gamma;\Delta \vdash e' \downarrow C$.

  (i) holds by reflexivity of $\equiv_{\mathsf{let}}$. Since $e' = e$ is a term in the left tridirectional system it contains no **let**s at all, so (ii) is trivially satisfied.

- **Case** $\bot\mathbb{L}$: Similar to the contra case. □

Theorem 5.80 (Let-Normal Completeness). *If $\mathcal{D}$ derives $\cdot;\cdot \vdash e \downarrow^{\mathbb{L}} C$ and $e \hookrightarrow L + e^*$ then $\cdot;\cdot \vdash L$ **in** $e^* \downarrow^{\mathsf{let}} C$.*

Proof. By Lemma 5.79, $\cdot;\cdot \vdash e' \downarrow^{\mathsf{let}} C$ where $e' \equiv_{\mathsf{let}} e$ and e' is let-respecting.

By Lemma 5.78, $\cdot;\cdot \vdash L'$ **in** $e'^* \downarrow^{\mathsf{let}} C$ where $\hookleftarrow(e') \hookrightarrow L' + e'^*$. Now we just need to show that $L' = L$ and $e'^* = e^*$.

	$e' \equiv_{\mathsf{let}} e$	Given
	$\hookleftarrow(e') = \hookleftarrow(e)$	By Proposition 5.10
	$= e$	By Proposition 5.9 (e is direct style)
	$\hookleftarrow(e') \hookrightarrow L' + e'^*$	Above
	$e \hookrightarrow L' + e'^*$	Substituting
	$e \hookrightarrow L + e^*$	Given
	L' **in** $e'^* = L$ **in** e^*	By Proposition 5.36
	$\cdot;\cdot \vdash L'$ **in** $e'^* \downarrow^{\mathsf{let}} C$	Above
☞	$\cdot;\cdot \vdash L$ **in** $e^* \downarrow^{\mathsf{let}} C$	Substituting

□

5.8 Extension to full pattern matching

This chapter has used the simplistic form of pattern matching used in Chapters 2 and 3. Extending it to the full formalism in Chapter 4 appears straightforward. The key is the minimal interaction between pattern matching and the let-normal transform. In particular, in **case** e **of** ms no linear variables—including those bound to subterms of e—can appear free in ms, since nothing in ms is in evaluation position; consequently, the let-normal translation "restarts" itself in each case arm. Moreover, the rules in Chapter 4 leave the term e alone entirely, hidden behind a FORGETTYPE, so the changes to term typing in this chapter should not interfere with them. Finally, the property of principal synthesis of values (Lemma 5.3) depends on having no unknown variables in terms, which is guaranteed by the premise $\Gamma, \mathit{FPV}(p) \vdash e \ \mathsf{ok}$ in rule casearm (Figure 4.4)—a premise which was superfluous in that chapter.

5.9 Related work

The effects of transformation to continuation passing style on the precision of program analyses such as 0-CFA have been studied for some time [SF94]. The effect depends on the specific details of the CPS transform and the analysis done [DD01, PW03].

The "analysis" in our work is the process of bidirectional checking/synthesis. Our soundness and completeness results prove that our let-normal transformation does not affect our analysis. It is not clear whether our result means anything for more traditional let-normal transformations and compiler analyses.

5.10 Conclusion

We now have a type system without obvious major impediments to implementation. The system is sound *and complete* with respect to the type assignment system (Chapter 2), in contrast to previous work in which completeness is lost; the tridirectional rule *can* be made into something practical.

Since we proved the safety of the type assignment system, the chain of soundness results (Figure 5.10) guarantee that if we run a program e whose let-normal translation typechecks in the system in this chapter, it will not go wrong.

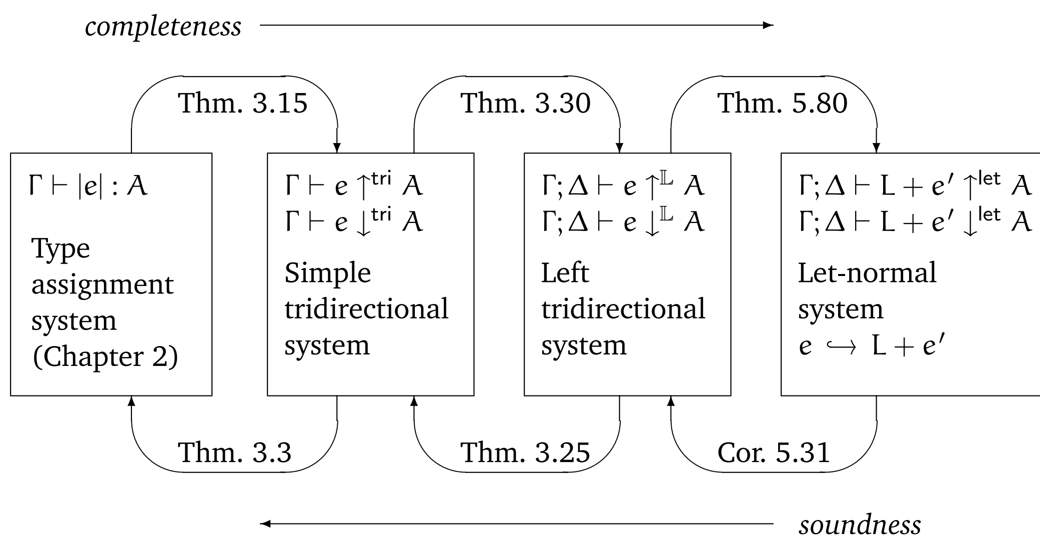


Figure 5.10: The menagerie: our type systems and the key results relating them

Chapter 6

Implementation

We have developed a typechecker, called Stardust, for a language and type system similar to that of Chapter 5, adapted to a subset of core Standard ML [MTHM97]. After describing the implemented language in more detail, we discuss how predefined identifiers (the 'basis') and user refinements are declared. We then give an overview of the implementation itself (Section 6.2), showing how certain impractical rules from the formal type system are elaborated into a practical constraint-based system and why incremental (online) constraint solving is needed. Next, we examine the interaction between the typechecker and the external constraint solver in detail, including how we handle index domains not supported by the external solver. Finally, we discuss the efficiency of typechecking and explain why we expect the system to scale well with larger programs.

6.1 The implemented language

A program in the implemented language consists of a sequence of SML datatype declarations—each of which may also declare a datasort relation, an index sort refining the datatype, and the refined types $\mathcal{S}(\mathsf{c})$ of the constructors—followed by one or more blocks. A *block* is a sequence of mutually recursive declarations (either **fun** ... **and** ... **and** ..., or just a single **fun** or **val** binding). Each block may be preceded by a type annotation of the form `(*[ ... ]*)` (the notation is adapted from Davies [Dav05a]). This annotation form appears as a comment `(*...*)` to Standard ML compilers, allowing programs in the subset language to be compiled normally. For example, in the following fragment the refined type annotation badLeft → rbt is processed only by Stardust.

```
(*[ val restore_left : badLeft → rbt
]*)
fun restore_left arg = case arg of
  ...
```

Contextual typing annotations can of course appear in nested declarations, but are not permitted on top-level declarations. This ensures that typechecking is modular, in the sense that each top-level block of mutually recursive declarations can be checked independently of the others.

The following are among the SML features not supported:

– parametric polymorphism

– modules
– **ref**
– user exception declarations
– records
– various forms of syntactic sugar, such as clausal function declarations and user-defined infix operators

However, Stardust does support all of the SML pattern forms (except literal patterns). It also supports exceptions through the usual **raise** and **handle** constructs, but the exception datatype is declared in the basis declaration file lib_basis.rml and cannot be supplemented by the user program.[1]

We give part of the grammar for the language in Figures 6.1 and 6.2, presented in an extended BNF in which $[\ldots]$, $(\ldots)^*$, and $(\ldots)^+$ respectively denote zero or one, zero or more (Kleene closure), and one or more occurrences of the enclosed string of symbols. Nonterminals are written *nonterm*. Terminals with several lexemes, such as identifiers ***id***, appear in ***bold italic***, while keywords appear in **bold** and other terminals with only one lexeme appear as themselves: $\rightarrow$, `:`, `&`, etc.

sort ::= *atomic-sort* (`*` *atomic-sort*)* | **unit** | `{` ***id*** `:` *sort* `|` *proposition* `}`
atomic-sort ::= ***id*** | `(` *sort* `)`
proposition ::= *index-exp* | *proposition* **and** *proposition* | *proposition* `\/` *proposition*
texp ::= **unit** | `(` *texp* `)`
| ***id*** [`(` *index-exp* (`,` *index-exp*)* `)`]
| *texp* $\rightarrow$ *texp* | *texp* `*` *texp*
| *texp* `&` *texp* | *texp* `\/` *texp*
| `{` *proposition* `}` *texp* | `[` *proposition* `]` *texp*
| `-`**all** ***id*** (`,` ***id***)* `:` *sort*`-` *texp* | `-`**exists** ***id*** (`,` ***id***)* `:` *sort*`-` *texp*
| **top** | **bot**
index-exp ::= ***id*** | ***id*** *index-exp* | *index-aexp*
index-aexp ::= `()` | `#` ***integer-literal*** `(` *index-exp* `)` | ***integer-literal*** | `(` *index-exp* (`,` *index-exp*)* `)`

Figure 6.1: Concrete syntax for types, sorts, propositions, and index expressions in Stardust

6.1.1 Type expressions

In addition to a subset of SML types, Stardust supports the following forms of type expression *texp* (corresponding to the formal system's A, B, etc.; see Figure 2.9):

[1] For historical reasons (p. 196), Stardust source files have the suffix `.rml`.

- atomic refinements, written ***id***(*index-exp*) where ***id*** is a datasort and *index-exp* is an index expression, discussed below (the index expression may be omitted, in which case the "injection" phase described in Section 6.3.2 will add a default index expression or an existential quantifier);
- intersection and union types, written *texp* & *texp* and *texp* \/ *texp*;
- guarded and asserting types, written {*proposition*} *texp* and [*proposition*] *texp*, respectively;
- index-level quantifiers, written -**all** ***id*** (, ***id***)* : *sort*- *texp* and -**exists** ***id*** (, ***id***)* : *sort*- *texp*;
- top and bottom types **top** and **bot**.

The following table relates the notation used in the rest of the thesis to the Stardust syntax.[2]

index expression	i	*index-exp*
index sort	γ	*sort*
atomic refinement	$\delta(i)$	***id***(*index-exp*)
... datasort only	n/a	***id***
intersection	$A_1 \wedge A_2$	*texp*$_1$ & *texp*$_2$
union	$A_1 \vee A_2$	*texp*$_1$ \/ *texp*$_2$
guarded type	$P \supset A$	{*proposition*} *texp*
asserting type	$P \otimes A$	[*proposition*] *texp*
universal quantifier	$\Pi a{:}\gamma.\,A$	-**all** ***id*** (, ***id***)* : *sort*- *texp*
existential quantifier	$\Sigma a{:}\gamma.\,A$	-**exists** ***id*** (, ***id***)* : *sort*- *texp*
greatest type	$\top$	**top**
empty type	$\bot$	**bot**

The grammar of type expressions in Figure 6.1 is simplified and ambiguous, so we need to specify precedence of type constructors:

(lowest) & $\rightarrow$ * \/ *(highest)*

The unary type constructors -**all** ...- and {P} A, which form definite types, have lower precedence than &, which is the corresponding definite binary constructor. Thus one can write types such as $\Pi a{:}\mathcal{Z}.\,(A \rightarrow \mathsf{list}(a)) \wedge (B \rightarrow \mathsf{list}(a+1))$ without parentheses, with a in scope in both parts of the intersection:

-**all** `a:int-` $A \rightarrow$ `list(a)` & $B \rightarrow$ `list(a+1)`

Likewise, -**exists** ...- and [P] A, which form indefinite types, have lower precedence than \/. However, they also have higher precedence than $\rightarrow$ (but lower than *). Unions, existentials, and asserting types usually (if not always) appear *within* arrows; typical index refinements of $A \rightarrow B$ are $\Pi a{:}\gamma_1.\,A(a) \rightarrow \Sigma b{:}\gamma_2.\,B(a,b)$ and $A \rightarrow B_1 \vee B_2$. We have not encountered unions of arrows or existential quantifications of arrows. Such types are perfectly legal in our formal systems, and are permitted by Stardust; the user just has to write enough parentheses.

[2]The effect of a type refined only by a datasort can be achieved in the formal systems by refining the type with a unit index sort and writing the unit index, or with an integer index sort and an arbitrary integer such as 0.

$$
\begin{aligned}
\textit{program} &::= (\textit{anno-datatype-dec})^* \ \texttt{;} \ \textit{blocks} \\
\textit{anno-datatype-dec} &::= [\textit{datatype-annos}] \ \textbf{datatype} \ \textit{datatype-dec} \ (\textbf{and} \ \textit{datatype-dec})^* \\
\textit{datatype-annos} &::= \texttt{(*[} \ (\textit{datatype-anno})^* \ \texttt{]*)} \\
\textit{datatype-anno} &::= \textbf{datacon} \ \boldsymbol{id} \ \texttt{:} \ \textit{texp} \mid \textbf{datatype} \ \boldsymbol{id} \ \textit{index-spec} \\
&\quad \mid \textbf{datasort} \ \boldsymbol{id} \ \texttt{:} \ \textit{datasort-pair} \ (\texttt{;} \ \textit{datasort-pair})^* \\
&\quad \mid \textbf{indexsort} \ \boldsymbol{id} \ \texttt{=} \ \textit{sort} \mid \textbf{indexfun} \ \boldsymbol{id} \ \texttt{:} \ \textit{sort} \rightarrow \textit{sort} \ (\texttt{,} \ \textit{sort} \rightarrow \textit{sort})^* \\
&\quad \mid \textbf{indexconstant} \ \boldsymbol{id} \ \texttt{:} \ \textit{sort} \mid \textbf{indexpred} \ \boldsymbol{id} \ [\texttt{:!} \ \boldsymbol{id}] \ \texttt{:} \ \textit{sort} \\
&\quad \mid \textbf{primitive type} \ \boldsymbol{id} \ \textit{index-spec} \\
&\quad \mid \textbf{primitive} \ (\textbf{fun} \mid \textbf{val}) \ \boldsymbol{id} \ \texttt{:} \ \textit{texp} \\
\textit{datasort-pair} &::= \boldsymbol{id} \ \texttt{<} \ \boldsymbol{id} \\
\textit{datatype-dec} &::= \boldsymbol{id} \ \texttt{=} \ \textit{con} \ (\ \texttt{|} \ \textit{con})^* \\
\textit{con} &::= \boldsymbol{id} \mid \boldsymbol{id} \ \textbf{of} \ \textit{texp} \ (\texttt{*} \ \textit{texp})^* \\
\textit{index-spec} &::= [\textbf{with} \ \textit{sort} \ [\textit{default-index-spec}]] \\
\textit{default-index-spec} &::= \texttt{=} \ \textit{index-exp}
\end{aligned}
$$

$$
\begin{aligned}
\textit{typedec} &::= \boldsymbol{id} \ \texttt{:} \ \textit{ctxanno} \mid \boldsymbol{id} \ \texttt{:!} \ \textit{texp} \\
\textit{ctxanno} &::= \textit{typing}(\texttt{,} \ \textit{typing})^* \\
\textit{typing} &::= \textit{texp} \mid \texttt{(} \ (\textit{ccelem})^* \Rightarrow \textit{texp} \ \texttt{)} \\
\textit{ccelem} &::= \boldsymbol{id} \ \texttt{::} \ \textit{sort} \mid \boldsymbol{id} \ \texttt{:} \ \textit{texp}
\end{aligned}
$$

$$
\textit{decpragmas} ::= \texttt{(*[} \ (\textbf{val} \ \textit{typedec})^+ \ \texttt{]*)}
$$

$$
\textit{exp} ::= \texttt{...} \mid \textit{exp} \ \texttt{:} \ \textit{ctxanno} \mid \texttt{(*[} \ \textit{ctxanno} \ \texttt{:} \ \texttt{]*)} \ \textit{exp}
$$

Figure 6.2: Part of the concrete syntax for datatype annotations, datatype declarations, value annotations, and value declarations in Stardust

6.1.2 Basis

Certain basic declarations, including the (refined) types of a small subset of the SML top-level Basis, are given in a file lib_basis.rml, which may be regarded as a header or "prelude": when a source file is given to Stardust, lib_basis.rml is processed first (in the same way as the main source file). In theory, lib_basis.rml is just another source file and could contain **val** and **fun** declarations; in practice, the *blocks* part of the *program* production (Figure 6.2) is empty.

lib_basis.rml is shown in Listing 6.1. The file begins with declarations of index constants, index functions, and index predicates. These declarations could logically be "hard-wired" in the typechecker (as the base sorts `int`, etc. are), but editing lib_basis.rml is easier than editing the typechecker itself. Note that **indexfun** declarations permit several function sorts, enabling a limited form of overloading, so that functions such as $*$ can be used in index expressions with both integers (e.g. `2*a`) and dimensions (e.g. `M*S`):

```
indexfun * : int * int → int, dim * dim → dim
```

We use a comma-separated list rather than writing `&`, because the latter would promise too much: we do not have general "intersection sorts".[3]

Listing 6.1: lib_basis.rml

```
(* The "basis" "library" *)

(*[
  (* ———————————— Index domain ———————————— *)

  indexconstant  false : bool
  indexconstant  true : bool
  indexconstant  NODIM : dim   (* The only place NODIM should need to be written
                                   is in this file, as the default index for type
                                   'real', so its ugliness is not a problem. *)
  indexconstant  M : dim
  indexconstant  S : dim
  indexconstant  KG : dim

  (* Index functions and predicates *)
  indexfun + : int * int → int
  indexfun - : int * int → int
  indexfun * : int * int → int, dim * dim → dim
  indexfun / : int * int → int, dim * dim → dim
  indexfun ^ : dim * int → dim
  indexfun mod : int * int → int

  indexpred = : int * int, bool * bool, dim * dim
  indexpred < : int * int
  indexpred <= : int * int
```

[3] These pseudo-"intersection sorts" are analogous to Standard ML's operator overloading mechanism, which effectively allows intersections (of arrows) on predefined identifiers such as the (ordinary, term-level) functions + and -, but does not allow intersections on user-declared functions: declaring **fun** *plus* $(x, y) = x + y$ yields *plus* : `int * int → int`, not *plus* : `int * int → int & real * real → real & ...`.

```
indexpred > :! <= : int * int
indexpred >= :! < : int * int
indexpred <> :! = : int * int, bool * bool, dim * dim

(* ———————————— Index refinements of primitive types ———————————— *)
primitive  type int with int
primitive  type real with dim = NODIM
primitive  type string

(* ———————————— Index refinement of 'bool' ———————————— *)
datacon false : bool(false)
datacon true : bool(true)
datatype  bool with bool

(* ———————————— Types of primitive values ———————————— *)
primitive  val M : real(M)
primitive  val S : real(S)
primitive  val KG : real(KG)

(* ———————————— Types of primitive functions ———————————— *)
primitive  fun ^: string * string → string
primitive  fun Int.toString : int → string

primitive  fun + : (-all a, b : int- int(a) * int(b) → int(a + b))
                 & (-all d : dim- real(d) * real(d) → real(d))
primitive  fun - : (-all a, b : int- int(a) * int(b) → int (a - b))
                 & (-all d : dim- real(d) * real(d) → real(d))

primitive  fun * :  (-all a, b : int- int(a) * int(b) → int(a * b))
                 & (-all a, b : int- {a >= 0}{b >= 0} int(a) * int(b)
                                → -exists c : int- [c >= 0] int(c))
                 & (-all d1, d2 : dim- real(d1) * real(d2) → real(d1 * d2))

primitive  fun div : int * int → int   (* -all a,b : int- int(a) * int(b) → int(a div b) *)
primitive  fun / : -all d1, d2 : dim- real(d1) * real(d2) → real(d1 / d2)
primitive  fun rem : int * int → int
primitive  fun mod : int * int → int   (* -all a,b : int- int(a) * int(b) → int(a mod b) *)
primitive  fun ~  : (-all a : int- int(a) → int(0 - a))
                  & (-all d : dim- real(d) → real(d))
primitive  fun abs : (-all a : int- int(a) → -exists b : int- [b >= 0] int(b))
                  & (-all d : dim- real(d) → real(d))
primitive  fun <  : (-all a, b : int- int(a) * int(b) → bool(a < b))
                  & (-all d : dim- real(d) * real(d) → bool)
primitive  fun <= : (-all a, b : int- int(a) * int(b) → bool(a <= b))
                  & (-all d : dim- real(d) * real(d) → bool)
primitive  fun >  : (-all a, b : int- int(a) * int(b) → bool(a > b))
                  & (-all d : dim- real(d) * real(d) → bool)
primitive  fun >= : (-all a, b : int- int(a) * int(b) → bool(a >= b))
                  & (-all d : dim- real(d) * real(d) → bool)
```

```
  primitive  fun =  : (-all  a, b : int- int(a) * int(b) → bool(a = b))
                    & (-all  a, b : bool- bool(a) * bool(b) → bool(a = b))
                    & (-all  d : dim- real(d) * real(d) → bool)
  primitive  fun <> : -all a, b : int- int(a) * int(b) → bool(a <> b)

  primitive  fun print : string → unit
  primitive  fun exit : unit → bot

  (* ———————————— Subset sorts ———————————— *)
  indexsort  nat = {a:int | a >= 0}
]*)

datatype  bool = false | true

datatype  exn = Match | Option | Subscript | Exn | Item_already_exists | NotFound
            | BadInput of string
;
```

Listing 6.1: lib_basis.rml

Lines starting with **indexpred** declare index predicates. To negate propositions, required for the Boolean sort (Section 6.6), the typechecker needs to know which predicates are complementary, such as > and <=: the former holds iff the latter does not. These are declared by an obscure form of **indexpred** declaration; the following line declares an index predicate > with domain $\mathcal{Z} * \mathcal{Z}$ and asserts that <= is its complement.

```
  indexpred  > :! <= : int * int
```

Primitive types such as int are declared next. An index refinement is optional; if present, a default index is optional. The following line gives real the index refinement dim, with default index NODIM (written *1* in Section 7.4).

```
  primitive  type real with dim = NODIM
```

Primitive constants and functions are declared with **primitive val** and **primitive fun**[4].

The last part of the lib_basis.rml annotation section declares subset sorts; there is presently only one, nat (corresponding to $\mathcal{N}$):

```
  (* ———————————— Subset sorts ———————————— *)
  indexsort  nat = {a:int | a >= 0}
```

lib_basis.rml concludes with datatype declarations. These are unremarkable except that the exn type is distinguished: the typechecker looks for a datatype with that name so it can check **raise** and **handle** expressions. As mentioned, Stardust does not support **exception** declarations.

6.1.3 Declaring refinements

Each program begins with zero or more blocks of datatype declarations, where a *block* is a collection of mutually recursive datatype declarations (in the rest of the program, a block is a collection

[4]**val** and **fun** are synonyms here; allowing only **val** would be more consistent with other annotation syntax.

of mutually recursive **fun/val** declarations). Each block can begin with a bracketed refinement declaration, such as the following comprised of **datacon** declarations and an index sort specification **datatype** list **with** int indicating that list is to be refined by the index sort of integers.

```
(*[
  datacon Nil : list(0)
  datacon Cons : -all u:int- int * list(u) → list(u + 1)
  datatype list with int
]*)
datatype list = Nil | Cons of int * list
```

The **datacon** declarations can be read as

$$\mathcal{S}(\mathsf{Nil}) = \mathsf{list}(0)$$
$$\mathcal{S}(\mathsf{Cons}) = \Pi u{:}\mathcal{Z}.\, \mathsf{int} * \mathsf{list}(u) \rightarrow \mathsf{list}(u+1)$$

Note that the implementation, unlike the formal system, supports nullary constructors.

To specify a datasort refinement, the user writes a "kernel" of the datasort relation $\preceq$ as a set of pairs, of which the system takes the reflexive-transitive closure. In addition to being more convenient for the user than writing the whole relation, this ensures reflexivity and transitivity of subsorting, needed for reflexivity and transitivity of subtyping. The declaration **datasort** $\mathsf{list} : \mathsf{odd} < \mathsf{list}; \mathsf{even} < \mathsf{list}$ defines the kernel of the datasort refinement of list to be $\{\langle \mathsf{odd}, \mathsf{list}\rangle, \langle \mathsf{even}, \mathsf{list}\rangle\}$; taking the reflexive-transitive closure yields

$$\{\langle \mathsf{odd}, \mathsf{list}\rangle, \langle \mathsf{even}, \mathsf{list}\rangle, \langle \mathsf{odd}, \mathsf{odd}\rangle, \langle \mathsf{even}, \mathsf{even}\rangle, \langle \mathsf{list}, \mathsf{list}\rangle\}$$

```
(*[
  datasort list : odd < list; even < list
  datacon Nil : even
  datacon Cons : int * odd → even
               & int * even → odd
               & int * list → list
]*)
datatype list = Nil | Cons of int * list
```

To refine by both a datasort and an index, the user can write the appropriate declarations:

```
(*[
  datasort list : odd < list; even < list
  datacon Nil : even(0)
  datacon Cons : -all u:int- int * odd(u) → even(u+1)
                           & int * even(u) → odd(u+1)
                           & int * list(u) → list(u+1)
]*)
datatype list = Nil | Cons of int * list
```

Remark 6.1. The use of $<$ in the concrete syntax is somewhat unfortunate. $\delta_1 < \delta_2$ suggests that δ_1 is *strictly* a subsort of δ_2, which need not be the case: $\delta_1 < \delta_2; \delta_2 < \delta_1$ is legal and creates

two synonymous datasorts: $\delta_1 \preceq \delta_2$ and $\delta_2 \preceq \delta_1$. The user almost certainly *intends* strict ordering (otherwise, why have both δ_1 and δ_2?), but it is not clear if that makes the syntax a lesser or greater misfortune.

6.1.4 Annotating declarations

Type annotations can be written in `(*[ ... ]*)` blocks, e.g.

(*[**val** *square* : -**all** d : dim- real(d) → real(d ∗ d)]*)
fun *square* x = x ∗ x

The relevant part of the grammar appears near the end of Figure 6.2. A *typedec* has either the form ***id*** : *ctxanno* or the unusual form ***id*** :! *texp*, which declares that the specified typing does *not* hold; this is primarily useful for testing purposes: typechecking "succeeds" if the system catches bugs in functions annotated with :!. A *ctxanno* is a comma-separated list of *typing*s, each of which can be just a *texp* (sugar for (⇒ *texp*)), or a genuine typing with a context: ((*ccelem*)* ⇒ *texp*). A *ccelem* (for *concrete context element*) is either an index variable typing ***id*** :: *sort* or a program variable typing ***id*** : *texp*.[5]

$$\begin{aligned} \textit{typing} &::= \textit{texp} \mid \texttt{(}\ (\textit{ccelem})^* \Rightarrow \textit{texp}\ \texttt{)} \\ \textit{ccelem} &::= \textbf{\textit{id}}\ \texttt{::}\ \textit{sort} \mid \textbf{\textit{id}}\ \texttt{:}\ \textit{texp} \end{aligned}$$

6.1.5 Annotating expressions

Expression annotations can be written as *exp* : *ctxanno*. Unfortunately, some such annotations are not valid SML, meaning that programs containing them cannot be compiled normally. For example, `double : int→int & real→real` is not a valid SML expression because SML does not have intersection types.

The best course would probably be to generate an SML program with such annotations removed or converted to the corresponding simple type, either as a byproduct of successful refinement type-checking or via a separate tool. In the interests of expediency, we instead simply provide a second, rather inelegant, syntax for annotated expressions:

(*[*ctxanno* :]*) *exp*

Annotations in this form, such as `(*[ int→int & real→real : ]*) double`, can be used without interfering with SML compilation, since the annotation is interpreted as a comment.

[5] Using `::` for index variables disambiguates strings such as `a:int`, which could otherwise refer either to an index variable of sort `int` (that is, $\mathcal{Z}$) or to a program variable of type int. We would like to allow index variable declarations to be elided entirely; it should be possible to infer them from the rest of the concrete context, e.g. the sort of a from x:list(a). Then there would be no need for the tedious `::`/`:` distinction.

6.1.6 Expression syntax

Besides the annotations discussed in the previous section, Stardust diverges from SML's surface syntax in a number of ways:

- the clausal form of function declarations is not allowed: the formal arguments to a function are written only once;
- the formal arguments must be a sequence (allowing curried style) of identifiers or tuples: **fun** *append* $(xs, ys) = \ldots$ is allowed, as are **fun** *map* $f\ xs = \ldots$ and even **fun** *composeMap* $(f, g)\ xs = \ldots$, but **fun** *map* $(\mathsf{Cons}(x, xs)) = \ldots$ has a constructor pattern and is not allowed;
- the only pattern permitted on the left hand side of a **val** binding **val** $p = e$ is a single variable; bindings such as **let val** $\mathsf{Cons}(x, \mathsf{Nil}) = e_1$ **in** e_2 **end** must be rewritten as **case** e_1 **of** $\mathsf{Cons}(x, \mathsf{Nil}) \Rightarrow e_2$;
- user-defined infix operators (including their precedence and associativity) are not supported—instead, the only infix identifiers are those in the Standard ML basis, and they have the corresponding precedence and associativity [MTHM97, p. 72];
- records are not supported;
- since modules are not supported, "`.`" is allowed in identifier names so that SML library functions like `Int.toString` can be used (a side effect is that non-SML code such as `val x.yz = 3` is accepted by Stardust, but this is only mildly unfortunate since such code will be rejected by SML compilers).

6.2 Design

Stardust consists of a parser, a few preprocessing phases, a translator from the source language to let-normal form, and a typechecker that includes interfaces to external constraint solvers. Parsing is unremarkable, and let-normal translation is straightforward except for minor issues arising from the additional complexity of the implemented language compared to the one in Chapter 5; see Section 6.3.5.

The type system in Chapter 5 presents several implementation challenges. The first is evident in rules such as ΠE, which pretend that we can somehow guess how to instantiate index variables. The usual approach, which we follow, is to postpone instantiation by generating constraints with existential variables. However, for efficient typechecking, constraint solving must be online. Otherwise, if we check $f(x)$ where $f : A \wedge B$, we may choose $f \uparrow A$, continue typechecking to the end of the block[6], find that the constraint is false, backtrack and choose $f \uparrow B$, etc. If $A = \mathsf{list}(0) \to A'$ and $x : \mathsf{list}(1)$, we should know immediately that trying $f \uparrow A$ is wrong, since $0 \doteq 1$ (the constraint generated by $\mathsf{list}(1) \leq \mathsf{list}(0)$) is invalid. Thus, we give the additional constraint to the solver on the fly, and when it reports that $0 \doteq 1$ is invalid we can proceed immediately to consider $f : B$. For

[6] We emphasize, again, that each top-level block of mutually recursive declarations (which may be a single function or **val** declaration) is typechecked independently.

n such choices, if the program is ill typed and all choices (would) ultimately fail, this takes us from typechecking the block 2^n times to only n times.

The constraint-based rules that the typechecker implements are largely unwritten; we show a few instances here. Writing Ω for solver contexts and Φ for the built-up constraint, the basic judgment form $\Gamma; \Delta \vdash e \downarrow A$ becomes $\Gamma; \Delta; \Omega; \Phi \vdash e \downarrow A \rightsquigarrow \Gamma'; \Omega'; \Phi'$ (resp. for $\uparrow$); the output Γ' lets us add existential index variables $\widehat{a}$. For example, rule ΠE becomes

$$\frac{\Gamma; \Delta \vdash e \uparrow \Pi a{:}\gamma.\, A \qquad \overline{\Gamma} \vdash i : \gamma}{\Gamma; \Delta \vdash e \uparrow [i/a]\, A}\,\Pi\mathrm{E} \qquad \Longrightarrow \qquad \frac{\Gamma; \Delta; \Omega; \Phi \vdash e \uparrow \Pi a{:}\gamma.\, A \rightsquigarrow (\Gamma'; \Omega'; \Phi')}{\Gamma; \Delta; \Omega; \Phi \vdash e \uparrow [\widehat{a}/a]\, A \rightsquigarrow (\Gamma', \widehat{a}{:}\gamma; \Omega'; \Phi')}$$

The constraint version of var does nothing interesting:

$$\frac{\Gamma(x) = A}{\Gamma; \cdot \vdash x \uparrow A}\,\mathsf{var} \qquad \Longrightarrow \qquad \frac{\Gamma(x) = A}{\Gamma; \cdot; \Omega; \Phi \vdash x \uparrow A \rightsquigarrow (\Gamma; \Omega; \Phi)}$$

The constraint version of $\supset\mathbb{L}$ adds a proposition to Φ:

$$\frac{\overline{\Gamma} \models P \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}(P \supset A) \vdash e \downarrow C}\,\supset\mathbb{L} \qquad \Longrightarrow \qquad \frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A; \Omega; \Phi \wedge P \vdash e \downarrow C \rightsquigarrow (\Gamma'; \Omega'; \Phi')}{\Gamma; \Delta; \Omega; \Phi \vdash e \downarrow C \rightsquigarrow (\Gamma'; \Omega'; \Phi')}$$

The typechecker is written in continuation-passing style; functions *check* and *infer* take continuations expecting $(\Gamma; \Omega; \Phi)$ and $(\Gamma; \Omega; \Phi; A)$ respectively, where A is the type synthesized by *infer*.

6.3 Initial phases

Stardust performs *sort checking* and applies several transformations to the program before let-translating and typechecking it. We briefly describe these initial phases.

6.3.1 Index sort checking

This phase (the only initial phase that does not transform the program) checks that index expressions are well sorted—rejecting, for example, $(a < b) + 1$ and $\#1(a)$ where $a : \mathcal{Z}$, as well as unknown identifiers. It also examines each type expression ***id***(i) and verifies that (1) ***id*** is the name of an indexed type (either a datatype or a refined primitive type such as int) or of a datasort refining such a type; (2) i is of the sort that indexes ***id***.[7]

6.3.2 Injection

For each type τ indexed by sort γ, the *injection* phase replaces occurrences of τ that are not indexed (i.e. do not have the form $\tau(i)$) with $\Sigma a{:}\gamma.\, \tau(a)$, unless a *default index* is specified in lib_basis.rml. For example, int is a primitive type indexed by the integer sort $\mathcal{Z}$, so injection replaces $\mathsf{int} * \mathsf{int}$ with $(\Sigma a_1{:}\mathcal{Z}.\, \mathsf{int}(a_1)) * (\Sigma a_2{:}\mathcal{Z}.\, \mathsf{int}(a_2))$. For type real, the default index is NODIM, as specified in lib_basis.rml:

```
primitive type real with dim = NODIM
```

Accordingly, injection replaces $\mathsf{real} * \mathsf{real}$ with $\mathsf{real}(\mathtt{NODIM}) * \mathsf{real}(\mathtt{NODIM})$.

[7]Our "index sort checking" should not be confused with Davies' "sort checking"—which we would call typechecking (of datasorts, not index sorts).

6.3.3 Product sort flattening

This phase replaces quantifiers $\Pi a{:}\gamma.\, A$ (respectively Σ) where γ is an n-ary product sort with

$$\Pi a_1{:}\gamma_1.\, \ldots \Pi a_n{:}\gamma_n.\, A$$

(resp. Σ) and substitutes $(a_1, \ldots, a_n)$ for a throughout A. (This leads to index equations on tuples, which are reduced pointwise to yield equations for the external constraint solver: $(i_1, i_2, i_3) \doteq (j_1, j_2, j_3)$ becomes $(i_1 \doteq j_1) \wedge (i_2 \doteq j_2) \wedge (i_3 \doteq j_3)$.)

6.3.4 Subset sort elimination

Most uses of subset sorts $\{a{:}\gamma \mid P\}$ can be eliminated very simply:

$$\begin{array}{lcl} \Pi a{:}\{a{:}\gamma \mid P\}.\, A & \text{becomes} & \Pi a{:}\gamma.\, (P \supset A) \\ \Sigma a{:}\{a{:}\gamma \mid P\}.\, A & \text{becomes} & \Sigma a{:}\gamma.\, (P \otimes A) \end{array}$$

For example, $\Pi a{:}\mathsf{nat}.$ is replaced by $\Pi a{:}\mathsf{int}.\, (a \geq 0) \supset A$. A more difficult situation is when a datatype is refined by a subset sort. Suppose we have the refinement declaration

$$\begin{array}{ll} (*[& \textbf{datacon}\ \mathsf{Nil} : \mathsf{list}(0) \\ & \textbf{datacon}\ \mathsf{Cons} : \Pi u{:}\mathsf{nat}.\, \mathsf{list}(u) \to \mathsf{list}(u+1) \\ & \textbf{datatype}\ \mathsf{list}\ \textbf{with}\ \mathsf{nat} \qquad\qquad]*) \end{array}$$

We must of course replace the quantifier in the declared type of Cons as above, yielding $\Pi u{:}\mathcal{Z}.\, (u \geq 0) \supset \mathsf{list}(u+1)$. We also need to replace each occurrence of the subset sort-indexed type, elsewhere in the program, with the appropriate asserting type; in general, if τ is a datatype indexed by a subset sort $\{a{:}\gamma \mid P\}$,

$$\tau(i) \quad \text{becomes} \quad [i/a]P \otimes \tau(i)$$

Typechecking will then ensure through rule $\otimes$I that $[i/a]P$ holds: to show $e \downarrow [i/a]P \otimes \tau(i)$ one must show $[i/a]P$.

However, the occurrences of $\tau(i)$ in the **datacon** declarations themselves cannot be replaced this way, because $\mathcal{S}(c) = (0 \geq 0) \otimes \mathsf{list}(0)$ is an indefinite type—not an A^{con}—which is not allowed (Section 2.4.4). In this case, the proposition $(0 \geq 0)$ is true, but if we had

$$\textbf{datacon}\ \mathsf{Nil} : \mathsf{list}(-1)$$

we would be able to create values $v : (-1 \geq 0) \otimes \mathsf{list}(-1)$ which, when used, would cause $-1 \geq 0$ to be added to Γ (rule $\otimes$E), allowing anything to typecheck (rule contra). It appears that we must validate the constructor types by checking the appropriate propositions (such as $-1 \geq 0$). We have not yet precisely formulated and implemented such a check, leaving it up to the user to verify that the indices in constructor types have the declared subset sort.

6.3.5 Let-normal translation

The let-normal translation is a fairly straightforward rendering of the rules in Figure 5.2, modulo syntactic differences between the formal language and the implemented one. Two points may be worth noting:

- The user-level **let val** $x = e_1$ **in** e_2 construct, being a sequence of subterms like $e_1 e_2$ and (e_1, e_2), requires a similar bifurcation on whether e_1 is a pre- or anti-value. Note that in practice e_1 is always a pre-value, since e_1 must be a synthesizing form for the **let** to typecheck (unless contra or a similar rule can be applied, but it is not likely that such a rule can be applied every time the **let** is visited during typechecking).

- Since not only pairs but n-ary tuples are allowed in the implemented language, the distinction on pre-/anti-values becomes more complex: given a tuple $(e_1, \ldots, e_n)$, if either none of the e_k are anti-values or only e_n is, the translation corresponds to the formal rule when e_1 is a pre-value, binding all synthesizing subterms outside the tuple; otherwise, only synthesizing subterms from the components up to and including the leftmost anti-value are bound outside. For example,

  $$(\check{e}_1, \widehat{e}_2, e_3, e_4) \quad \text{is translated to} \quad L_1, L_2 \;\textbf{in}\; (e_1', e_2', (L_3 \;\textbf{in}\; e_3'), (L_4 \;\textbf{in}\; e_4'))$$

  Here it does not matter whether e_3 or e_4 are pre- or anti-values, since the anti-value $\widehat{e}_2$ blocks any of their subterms from being in evaluation position outside the tuple. This corresponds to the formal rule for (e_1, e_2) in the case where e_1 is an anti-value; that rule does not care if e_2 is a pre- or anti-value.

6.4 Interface to an ideal constraint solver

We would like a constraint solver that supports the following for all index domains of interest:

1. A notion of *solver context* (represented by Ω) that encapsulates assumptions, corresponding to $\overline{\Gamma}$; we will abuse notation and write $\Omega \models P$ to mean that P is true under the assumptions *encapsulated by* Ω, etc.

2. An ASSERT operation taking a context Ω and proposition P, yielding one of three answers:

   (a) Valid if P is already valid under the current assumptions, i.e. $\Omega \models P$;

   (b) Invalid if P is unsatisfiable, that is, leads to an inconsistent set of assumptions: $\Omega, P \models \bot$;

   (c) Contingent(Ω') if $\Omega \not\models P$ and $\Omega, P \not\models \bot$; yields a new context $\Omega' = \Omega, P$.

3. A VALID operation taking a context Ω and proposition P, and returning one of two possible answers:

   (a) Valid if P is valid under the current assumptions, i.e. $\Omega \models P$;

   (b) Invalid otherwise.

4. A DECLARE operation taking a context Ω, name a, and index sort γ, which declares a to be a (universally quantified) variable of sort γ and returns the resulting context $\Omega' = \Omega, a{:}\gamma$.

Implicit in this specification is that the contexts Ω are persistent: if $\text{ASSERT}(\Omega_1, P)$ yields $\text{Contingent}(\Omega_2)$, the "earlier" context Ω_1 should remain unchanged. This is a key property, given all the backtracking the typechecker does. Where a constraint solver does not have this property, it can be simulated, though at some cost; see Section 6.11.1. Likewise, where the constraint solver does not support an index domain, propositions in that domain must be reduced to propositions in a supported domain.

6.5 Constraint-based typechecking

The *Typecheck* module (the typechecker proper) calls a *Solve* module, which in turn calls one of the following, based on a selection of the user (defaulting to *Cvcl*):

- *Cvcl* implements an interface to CVC Lite (Section 6.5.2) as a shared library;
- *CvclPiped* implements an interface to CVC Lite as a separate process, communicating via Unix pipes;
- *Ics* implements an interface to ICS (Section 6.5.1) as a separate process, communicating via Unix pipes.

Within *Solve* there is a notion of *state* that is independent of the particular constraint solver used.

```
datatype state = STATE of {assn : indexassn list,
                           constraint : constraint,
                           exiSub : ExiSub.exisub,
                           context : U.context}
```

The field `assn` is a list of assumptions corresponding to some $\overline{\Gamma}$ (Γ restricted to index variable sortings and propositions); `constraint` is a proposition[8]; `exiSub` is a substitution containing solutions for existentially quantified variables; and `context` is the *context* manipulated by the functor argument *U*, encapsulating the particular solver interface's state. Relative to the rules sketched above, `assn` corresponds to $\overline{\Gamma}$, `constraint` to Φ, and `context` to Ω. For all states, the assumptions encapsulated (on the external constraint solver) by `ics` (Ω) correspond to `assn` ($\overline{\Gamma}$).

Our constraint solvers do not support existential variables at all (ICS), or support them in a formally incomplete way (CVC Lite). This has significant repercussions on the whole design. The typechecker itself manages existentials, and lies to the constraint solver by telling it that existential variables are universal. Therefore, when adding a constraint we cannot immediately check its validity, since the constraint may include existential variables that the solver thinks are universal: we cannot directly check $\widehat{a}{:}\mathsf{int} \models \widehat{a} \doteq 0$ (meaning $\exists a.\ a = 0$), only $a{:}\mathsf{int} \models a \doteq 0$ (meaning $\forall a.\ a = 0$) with a universally quantified. Clearly, the first relation should hold and the second should not. Fortunately, we can still "fail early" (recalling $f : A \wedge B$ from the example earlier).

[8]We use P for both user *propositions*, which correspond to the propositions the user can write in the index domain, and *constraints*, which correspond to propositions built up internally. The latter are somewhat richer, including quantifiers $\forall a{:}\gamma.\ P$, $\exists \widehat{a}{:}\gamma.\ P$ and implications $P_1 \Rightarrow P_2$.

Instead of checking validity, we assert the new constraint, adding it to the assumptions. If the resulting assumptions are inconsistent (as with $0 \doteq 1$, or—less trivially—$a \doteq a+1$), no instantiation of existential variables can make the constraint valid, so we can correctly fail, and backtrack as needed.

Of course, we must check validity of the constraint at some point! Otherwise, given a constraint $b \doteq 0$, we would conclude $b{:}\mathsf{int} \models b \doteq 0$ since $b \doteq 0$ is a consistent assumption. Therefore, in addition to asserting $b \doteq 0$, we add it to a constraint built up in a manner similar to off-line constraint solving. Eventually, the typechecker solves for existentials and asks the solver if the built-up constraint is valid; see the description of *forceElimExistential* below.

The most important functions in *Solve* from the typechecker's perspective are *accumulate* and *assume*.

```
val accumulate : state * Indexing.constraint → state option
        (* returns NONE if inconsistent *)
val assume : state * Indexing.constraint → state option
        (* returns NONE if inconsistent *)
```

accumulate(s, P) conjoins the constraint P to the `constraint` in s. If P contains any existential variables, the constraint P is actually given to the constraint solver as an *assumption* (ASSERT); an inconsistent result means that P is unsatisfiable, in which case any further attempts to typecheck along this part of the search space are doomed to fail, even if some of the variables in P are existential. For example, if $\widehat{a}$ is an existential and $P = (\widehat{a} \doteq \widehat{a} + 1)$, *accumulate* returns `NONE`. If all variables in P are universally quantified, P is checked (VALID), and `NONE` is returned if P is not valid.

assume(s, P) also gives P to the constraint solver as an assumption, but adds P to the assumptions `assn` rather than to `constraint`. Like *accumulate*, *assume* returns `NONE` if the resulting assumptions would be inconsistent.

```
val forceElimExistential : state → Indexing.constraint → state * Indexing.constraint
```

The *forceElimExistential* function is called when the scope of an existentially quantified variable is closed; the variable must be eliminated before the accumulated constraint can be verified. The second argument must be a constraint of the form $\exists\widehat{a}.P$. The actual elimination is carried out by manipulating each equation $i \doteq j$ in P (where $\widehat{a} \in FV(i \doteq j)$) until the variable $\widehat{a}$ is isolated ($\widehat{a} \doteq j$), at which point j is substituted for $\widehat{a}$ throughout, in both the accumulated constraint (to be validated) and the assumptions (propositions in Γ). The manipulation follows the system of rewrite rules in Figure 6.3, assuming without loss of generality that $\widehat{a}$ is free on the left-hand side.

$$
\begin{array}{rcll}
u_1 + u_2 \doteq j & \longrightarrow & u_1 \doteq j - u_2 & \text{if } \widehat{a} \in FV(u_1) \\
u_1 + u_2 \doteq j & \longrightarrow & u_2 \doteq j - u_1 & \text{if } \widehat{a} \in FV(u_2) \\
u_1 - u_2 \doteq j & \longrightarrow & u_1 \doteq j + u_2 & \text{if } \widehat{a} \in FV(u_1) \\
u_1 - u_2 \doteq j & \longrightarrow & u_2 \doteq u_1 - j & \text{if } \widehat{a} \in FV(u_2) \\
u_1 * u_2 \doteq j & \longrightarrow & u_1 \doteq (1/u_2) * j & \text{if } \widehat{a} \in FV(u_1) \\
u_1 * u_2 \doteq j & \longrightarrow & u_2 \doteq (1/u_1) * j & \text{if } \widehat{a} \in FV(u_2)
\end{array}
$$

Figure 6.3: Existential elimination rules for integer equations (assuming $\widehat{a}$ free on the left-hand side)

If *forceElimExistential* finds a solution i for $\widehat{a}$, it returns $(s', [i/\widehat{a}]\, P)$ where s' is s with `exiSub` extended by $i/\widehat{a}$. If it fails, it returns $(s, \exists\widehat{a}.\, P)$, in which case typechecking will only succeed if P is valid even when $\widehat{a}$ is regarded as universal (for example, if $P = \widehat{a} \doteq \widehat{a}$, since $\forall a.\, (a \doteq a)$ is valid).

The last key function in *Solve* is *solve*, which checks if the accumulated constraint is valid; it is called by *Typecheck* when it reaches the end of a block.

```
val solve : state → bool
```

6.5.1 Interface to ICS

Stardust includes an interface to the ICS system [dMOR+04, SRI03] as an external constraint solver. ICS, developed at SRI, has cooperating decision procedures for fragments of rational arithmetic, uninterpreted functions, functional arrays, and other theories; the typechecker presently uses only the arithmetic theory. While there is a notion of "current context" in the ICS interface (for example, ICS's ASSERT operation takes only a proposition and implicitly uses the current context as the Ω), previously constructed contexts can be saved and restored quickly, yielding an interface extremely close to the idealized one presented above. This is no coincidence, since we designed the typechecker expecting that we would use ICS.

A significant specification gap is that ICS does not properly support Booleans. It has a SAT solver, but that does not cooperate with the rest of the system. The bit vector theory seemed promising, but encoding Booleans as bit vectors of length 1 yielded unsatisfying results: asserting $a = 1$ and $b \neq a$ does not make $b = 0$ valid! Fixing this might seem easy in principle but ICS is no longer supported (and SRI's oppressive licensing makes it difficult to do on our own).

We do not use ICS as a library; instead, it runs as a separate process and we communicate through Unix pipes.

6.5.2 Interface to CVC Lite

Stardust also includes an interface to CVC Lite [BB04, Bar06b], the successor to CVC, the Cooperating Validity Checker [SBD02], which in turn succeeded SVC, the Stanford Validity Checker [BDL96]. CVC Lite has cooperating decision procedures for fragments of integer and rational arithmetic, Boolean propositions (including conjunction, disjunction, negation, and implication), uninterpreted functions, bit vectors, and other theories; we presently use only the integer and Boolean theories. It has limited support for quantifiers, both universal and existential (free variables are, as in ICS, always universal); a response of Invalid may be given even when an existential solution exists. We have not explored whether that limited support is sufficient for Stardust; it could be as powerful than our home-grown existentials, and might lead to a simpler design.

Unlike ICS, CVC Lite does not support persistent contexts. We discuss the impact of this in Section 6.11.

CVC Lite has recently (2006) become CVC3 [Bar06a]. We hope to add support for CVC3; that should allow us to easily implement an index domain where the objects are inductive datatypes (see p. 239), since CVC3 supports them.

6.6 Internal index domains

Stardust supports index domains that the external solvers do not. Propositions in these "internal" domains must be nontrivially manipulated before being given to an external solver. Specifically:

- Equations in the dimension sort dim are reduced to propositions in the integer domain by a process described in the next chapter (Section 7.4.4).

- ICS does not support Booleans. When ICS is used (as opposed to CVC Lite), implications $P_1 \Rightarrow P_2$ are checked by asserting P_1 and checking validity of P_2. Note that implications are not "first class" since they cannot be asserted. They do not appear in the source language of propositions available to the user; instead, they result from the encapsulation of constraints[9]: when an assumption P is discharged, the generated constraint P' is replaced with $P \Rightarrow P'$. That constraint's validity may be checked, but it will not be asserted.

  A consequence of being unable to assert implications is that we cannot support "first class" negations just by translating $\neg P$ to $P \Rightarrow \mathsf{False}$, which in turn means that we cannot, in general, trivially support disjunctions (Section 6.9), at least when using ICS.

- When ICS is used, assertion of an index equation $i_1 \doteq i_2$ where i_1 and i_2 have sort bool is handled as follows:

$$\begin{array}{rcllcl} \mathsf{True} & \doteq & i_2 & & \longrightarrow & \text{assert } i_2 \\ \mathsf{False} & \doteq & i_2 & & \longrightarrow & \text{assert the negation of } i_2 \\ i_1 & \doteq & \mathsf{True} & & \longrightarrow & \text{assert } i_1 \\ i_2 & \doteq & \mathsf{False} & & \longrightarrow & \text{assert the negation of } i_1 \end{array}$$

  We would like to check the general case $i_1 \doteq i_2$ by transforming it into $(i_1 \Rightarrow i_2) \land (i_2 \Rightarrow i_1)$, but one cannot assert an implication in ICS. In this case, Stardust ignores the assertion and does not give it to ICS, which may slow typechecking: not knowing $i_1 \doteq i_2$ may lead to a failure to notice inconsistent assumptions. The more serious consequence comes in the final (which, in this case, is also the *first*) validity check. In the final validity check, to check that $\mathsf{True} \doteq i_2$ is valid we check that i_2 is valid, and so forth. Here we can usually check an equation of the form $i_1 \doteq i_2$ as if it were $(i_1 \Rightarrow i_2) \land (i_2 \Rightarrow i_1)$: assert i_1, check i_2, roll back the context, assert i_2, and check i_1. This assumes that i_1 and i_2 can be asserted, i.e. that they do not themselves include implications. In that case the antecedent is ignored, as above, but with more serious consequences: instead of a performance degradation, the power of the system is reduced, since equations that should be valid will not be. Again, none of these problems occur when CVC Lite is used, since it supports Booleans directly.

- Finally ICS does not properly support integers, only rationals, so we "stultify" certain propositions: for example, instead of asserting $a > b$ we assert $a \geq b + 1$, excluding the open interval $(b, b+1)$ that contains no integers.

[9]Implications also arise from equations in the Boolean sort, discussed in the next item.

6.7 Optimizations

We discuss several optimizations and whether they are performed: how we separate the synthesis judgment into two, memoization (and why experiments led us to abandon it), left rule optimizations, and slack variables.

6.7.1 Improvement of the synthesis judgment

For efficiency, the implementation separates the synthesis judgment into two. Suppose we have the term

$$\textbf{let}\ \bar{\mathbf{x}} = x\ \textbf{in}\ \textbf{let}\ \bar{\mathbf{y}} = y\ \textbf{in} \ldots \text{(several more } \textbf{let}\text{s)} \ldots$$

and $\Gamma(x) = A_1 \wedge A_2$. Applying rule let, we need to synthesize a type for x. At least three different derivations yield a judgment of the form $\ldots \vdash x \uparrow A$ for some A:

$$\dfrac{\Gamma(x) = A_1 \wedge A_2}{\Gamma \vdash x \uparrow A_1 \wedge A_2}\ \mathsf{var} \qquad \dfrac{\dfrac{\Gamma(x) = A_1 \wedge A_2}{\Gamma \vdash x \uparrow A_1 \wedge A_2}\ \mathsf{var}}{\Gamma \vdash x \uparrow A_1}\ {\wedge}\mathrm{E}_1 \qquad \dfrac{\dfrac{\Gamma(x) = A_1 \wedge A_2}{\Gamma \vdash x \uparrow A_1 \wedge A_2}\ \mathsf{var}}{\Gamma \vdash x \uparrow A_2}\ {\wedge}\mathrm{E}_2$$

It is best to derive the first of these, so that in the second premise of let we will have $\bar{\mathbf{x}}{:}A_1 \wedge A_2$, and wherever $\bar{\mathbf{x}}$ is eventually used, we will have the flexibility to apply either $\wedge\mathrm{E}_1$ or $\wedge\mathrm{E}_2$ at that point, as needed (for example, if we use rule $\to$E on a term $\bar{\mathbf{x}}\,\bar{\mathbf{y}}$). If we try $\bar{\mathbf{x}}{:}A_1 \wedge A_2$ and fail, we will also fail with $\bar{\mathbf{x}}{:}A_1$ and $\bar{\mathbf{x}}{:}A_2$: the second and third derivations cause considerable backtracking. We avoid such backtracking by introducing a $\uparrow_1$ judgment form that excludes non-invertible rules such as $\wedge\mathrm{E}_{1,2}$. Thus $\uparrow_1$ produces principally synthesizing types, such as $A_1 \wedge A_2$, from which A_1 and A_2 can be synthesized. (Principal synthesis also came up in Chapter 5; see Section 5.3.1.)

In contrast, if (again) $\Gamma(x) = A_1 \wedge A_2$ and we are trying to synthesize a type for $x(e)$ via rule $\to$E, we *must* apply $\wedge\mathrm{E}_1$ or $\wedge\mathrm{E}_2$ so we can get an arrow type for x. Likewise, we must eliminate intersections when applying $*\mathrm{E}_1$ or $*\mathrm{E}_2$ (to obtain a $*$ type) and in δE (to obtain a $\delta(i)$ type). We introduce a $\uparrow_2$ judgment form, equivalent to the original $\uparrow$. For example, there are versions of $\wedge\mathrm{E}_1$ and $\wedge\mathrm{E}_2$ concluding $\uparrow_2$, allowing the system to break down the property type $(A_1 \to A_2) \wedge (A_1' \to A_2')$ into its ordinary components $A_1 \to A_2$ and $A_1' \to A_2'$.

Relation to focusing

This technique is related to focusing in linear logic [And92]. Consider the structure of a derivation involving direct$\mathbb{L}$:

$$\dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Gamma \vdash e' \uparrow A & \Gamma, \bar{\mathbf{x}}{:}A \vdash \mathcal{E}[\bar{\mathbf{x}}] \downarrow C\end{array}}{\Gamma \vdash \mathcal{E}[e'] \downarrow C}\ \mathsf{direct}\mathbb{L}$$

By constructing $\mathcal{D}_1$, a type A for e' is found; during construction of $\mathcal{D}_2$, the type A appears as part of an assumption, to the left of the turnstile. The construction of $\mathcal{D}_1$ corresponds to the

inversion phase, in which one does not apply non-invertible rules. The part of $\mathcal{D}_2$ in which $\bar{\mathbf{x}}$ is used corresponds to the focusing phase, in which non-invertible rules may be applied. To see this correspondence, we must know which linear logic sequent rule corresponds to a given synthesis rule.

Operationally, synthesis rules proceed from a known fact about the subject term to another fact. For instance, $\wedge E_1$ proceeds from the fact that e has the property expressed by $A_1 \wedge A_2$ to the fact that e has the property expressed by A_1. This is analogous to a left rule in the sequent calculus, namely the additive conjunction rule $\& L_1$ (as well as our own left rule $\wedge\mathbb{L}_1$):

$$\frac{\ldots, A_1 \vdash C}{\ldots, A_1 \& A_2 \vdash C}\ \& L_1$$

Asynchronous rules are applied during the inversion phase, *synchronous* rules in the focusing phase. To know when to permit application of the left rule $\& L_1$, we simply need to ask whether $\&$ is a left synchronous or left asynchronous connective. It is left synchronous, so the left rule $\& L_1$ should not be applied in the inversion phase.

Thus, $\wedge$ has the character of a left synchronous connective. In contrast, $\rightarrow$ is left asynchronous, with no clear correspondence to linear logic; of course, $\rightarrow$ is not a property type and $\rightarrow$E decomposes its subject term, so a direct correspondence would be startling. On the other hand, an indefinite property type such as $\vee$ should correspond to additive disjunction $\oplus$, and indeed it would seem that *if* we had any $\vee$ rules with a synthesized conclusion, they could be safely applied in the inversion phase.

Implementation

The two judgment forms are implemented through an additional argument to *infer*:

```
datatype inference_disposition = MAINTAIN_PRINCIPALITY | WANT_ORDINARY_TYPE
```

If the $\uparrow_1$ form is to be derived, MAINTAIN_PRINCIPALITY is passed to *infer*, whereas for the $\uparrow_2$ form WANT_ORDINARY_TYPE is passed.

6.7.2 Memoization

The backtracking due to union and intersection types, such as with rules $\wedge E_1$ and $\wedge E_2$, can be problematic. Consider a **case** expression with several arms. If every arm but the last typechecks, the failure in the last arm *could* be due to a choice made while checking an earlier arm: that choice could lead to a different generated constraint, potentially affecting the success of checking the final arm. Thus, if n two-way choices are made, we will traverse all paths on a complete binary tree of height n. However, the common case appears to be complete independence between the well-typedness of the arms. It would be incorrect to assume such independence, but we can detect some instances of independence through memoization [Mic68]. Essentially, when deriving $\Gamma; \Delta; \Omega; \Phi \vdash e \downarrow A \;\rightsquigarrow\; (\Gamma'; \Omega'; \Phi')$ we can memoize the result (a sequence of context/constraint pairs), keyed by $(\Gamma, \Delta, \Phi, e, A)$ (under α-equivalence of index variable contexts). The result is a

sequence since there may be several possible output pairs, corresponding to different choices made while checking e.

However, our experience indicates that memoization is a net loss. Few memoized results can actually be used. Even a minor and irrelevant change in the assumptions will cause a table miss, and the typechecker rarely needs to derive a judgment absolutely identical (modulo α-equivalence) to one already derived. A more robust notion of equivalence could be helpful here.

Note that these problems arise from the combination of intersection (and union) types with index refinements; Davies' system has no index refinements, so case arms are truly independent and memoization is highly beneficial [Dav05a, pp. 271–273]; in DML [Xi98], there are no intersections and no unions, and thus no need to backtrack.

6.7.3 Left rule optimizations

The left rules (Figure 3.8) are troublesome: they can be applied whenever we need to derive a checking judgment and have a linear variable of type $\bot$, Σ, $\vee$, $\otimes$, Π, $\wedge$ or $\supset$ somewhere in Δ. Fortunately, the rules for the indefinite types—$\bot\mathbb{L}$, $\Sigma\mathbb{L}$, $\vee\mathbb{L}$, $\otimes\mathbb{L}$—are invertible, so they can safely be applied as soon as a linear variable of the relevant type is added to the context. Moreover, the rules for the definite types ($\wedge\mathbb{L}_{1,2}$, etc.) need not be applied when doing so would only expose ordinary type constructors: there is no need to look inside the $\wedge$ in $\bar{\mathbf{x}}$:$(A_1 \to A_2) \wedge (B_1 \to B_2)$, for example, since it only exposes arrows, which have no left rule; when $\bar{\mathbf{x}}$ is actually used, the ordinary elimination rules like $\wedge E_1$ can be applied. However, if an indefinite type could be exposed, we may have to apply $\wedge\mathbb{L}_1$ or $\wedge\mathbb{L}_2$. This situation—a kind of quantifier alternation—is rare in the examples we have tried: one tends to see either various definite type constructors applied to ordinary types (e.g. a universally quantified intersection of arrows), or an indefinite type constructor applied to ordinary types (e.g. $\delta(i) \vee \delta(j)$). It does not matter if an ordinary type contains *nested* definite or indefinite property types, since those are not manipulated by the type system until the ordinary type is eliminated.

As with the synthesis-judgment technique above (Section 6.7.1), there is some connection to focusing in linear logic: the invertible decomposition of the indefinite type $\vee$ is similar to the left asynchronicity of $\oplus$, while the situation with the definite type $\wedge$ corresponds to the left synchronicity of $\&$.

6.7.4 Slack variables

We have not encountered many slack variable bindings. Annotated values—terms of the form $(v : As)$—are rare in our examples; most annotations are on **fun** declarations, which correspond to a **fix** of a λ (for mutually recursive declarations, a **fix** of a tuple of λs) in our formal systems, and we do not consider **fix** a value. We have not tried to assess how many annotated values appear in real SML programs, so it is possible that a redesign of the slack variable scheme would be necessary, but we have at least seen no discouraging evidence.

6.8 Pattern checking

Subtraction and intersection are implemented such that they should have the properties required in Chapter 4. The implementation follows the useful optimization described in Section 4.8. Rules $\wedge$-p, Σ-p and $\otimes$-p (p. 102) are not implemented, not for any deep reason but simply because we encountered no cases in which they were needed.

6.9 Disjunctions

Some programs call for case analysis (splitting) of disjunctive propositions. Consider the refined binary search trees in Figure 6.4, in which values of tree type are refined by pairs (l, u) of integer indices: l and u are the least and greatest keys appearing in the tree. Datasorts are used to distinguish empty (Leaf) and nonempty (Node) trees; by convention, both l and u are 0 for empty trees.

For example, $\mathsf{Node}(5, \mathsf{Node}(1, \mathsf{Leaf}, \mathsf{Leaf}), \mathsf{Node}(8, \mathsf{Leaf}, \mathsf{Leaf}))$ has type $\mathsf{tree}(1, 8)$. The refinement ensures that all trees are ordered—disordered trees such as $\mathsf{Node}(7, \mathsf{Leaf}, \mathsf{Node}(5, \mathsf{Leaf}, \mathsf{Leaf}))$ have no type, because $\mathsf{Node}(5, \mathsf{Leaf}, \mathsf{Leaf})$ has type $\mathsf{tree}(5, 5)$, but $\mathcal{S}(\mathsf{Node})$ requires that the first index of the right child is greater than or equal to the integer stored at that node (7). The constructor

```
(*[
  datatype tree with int * int
  datasort tree : empty < tree; nonempty < tree
  datacon Leaf : empty(0,0)
  datacon Node :
      (-all x,l1,u1,l2,u2 : int- {l1<=u1 and l2<=u2 and u1<=x and x<=l2}
           int(x) * nonempty(l1,u1) * nonempty(l2,u2) → nonempty(l1,u2))
    & (-all x,l2,u2 : int- {x<=l2 and l2<=u2}
           int(x) * empty(0,0) * nonempty(l2,u2) → nonempty(x,u2))
    & (-all x,l1,u1 : int- {l1<=u1 and u1<=x}
           int(x) * nonempty(l1,u1) * empty(0,0) → nonempty(l1,x))
    & (-all x : int-
           int(x) * empty(0,0) * empty(0,0) → nonempty(x,x))
]*)
datatype tree = Leaf | Node of int * tree * tree
```

Figure 6.4: Ordered binary search trees

types are rather straightforward (if somewhat clumsy since $\mathcal{S}(\mathsf{Node})$ enumerates all permutations of emptiness/nonemptiness in the children), but the obvious *insert* function raises some issues. Inserting a key into an empty tree is trivial, as shown in the first line of the annotation; when inserting a key into a nonempty tree we have to separately consider the cases where (1) the inserted key is smaller than the minimum key in the existing tree, (2) greater than the maximum key in the existing tree, (3) between the minimum and the maximum.

```
(*[ val insert : -all key:int- (empty(0,0) * int(key) → nonempty(key,key))
```

```
    & -all l,u : int-
        ({key<=l} nonempty(l,u) * int(key) → nonempty(key,u))
      & ({key>=u} nonempty(l,u) * int(key) → nonempty(l,key))
      & ({key>=l and key<=u} nonempty(l,u) * int(key) → nonempty(l,u))
]*)
```

The difficulty comes when *insert* is applied (whether recursively or by another function) and the new key's standing vis-à-vis the minimum and maximum keys in the existing tree is not known. In our type system, we cannot derive

$$\Gamma, l{:}\mathcal{Z}, u{:}\mathcal{Z}, t{:}\mathsf{tree}(l,u), a{:}\mathcal{Z}, x{:}\mathsf{int}(a) \vdash \mathit{insert}(t,x) \uparrow \dots$$

because we cannot show $a \leq l$ or $a \geq u$, needed to apply $\supset$E. We need to notice that the disjunction $(a \leq l) \vee (a > l)$ holds so we can split it, checking the term separately under $\Gamma, \dots, a \leq l$ and under $\Gamma, \dots, a > l$. (In the latter case, we need to likewise split $(a \geq u) \vee (a < u)$.)

At the level of the inference rules, we can simply add a disjunctive analogue of rule contra: as contra deals with falsehood (the disjunction of nothing), rule disjunction would deal with binary disjunctions.

$$\frac{\overline{\Gamma} \models \bot}{\Gamma \vdash e \downarrow C}\,\mathsf{contra} \qquad \frac{\overline{\Gamma} \models P_1 \vee P_2 \qquad \Gamma, P_1 \vdash e \downarrow C \qquad \Gamma, P_2 \vdash e \downarrow C}{\Gamma \vdash e \downarrow C}\,\mathsf{disjunction}$$

However, such a rule is harder to implement than contra. The premise $\overline{\Gamma} \models P_1 \vee P_2$ is nontrivial: P_1 and P_2 are not known and there is no obvious way to have the constraint solver report relevant disjunctions. This can be overcome by invoking the rule only when $P_1 \vee P_2$ is added to Γ, which occurs when the user writes annotations containing propositions $P_1 \vee P_2$, e.g. when checking a function against $((a \geq u) \vee (a < u)) \supset \dots$. Of course such a proposition always holds—its sole purpose is to signal to the typechecker that disjunction should be applied.

Remark 6.2 (Convexity and the disjunction rule). An index domain (or "theory") is *convex* if, whenever some disjunction of equations is entailed by assumptions[10], there is a *specific* equation that is also entailed [NO79, p. 252]. For instance, the linear integers are *not* convex:

$$a{:}\mathcal{Z}, a \geq 0, a \leq 2 \models (a \doteq 0) \vee (a \doteq 1) \vee (a \doteq 2)$$

but $a{:}\mathcal{Z}, a \geq 0, a \leq 2 \not\models a \doteq k$ for any $k \in \{0,1,2\}$.

In a convex theory, rule disjunction seems less useful, yet not superfluous. Case-analyzing the disjunction $(a < 0) \vee (a \geq 0)$ is just as useful in the rationals (convex) as the integers (nonconvex), but the only situation in which the disjunction $P = (a \doteq 0) \vee (a \doteq 1) \vee (a \doteq 2)$ is entailed in the rationals is when P itself is an assumption.

Validation of disjunctions is possible in many cases even if the constraint solver does not directly support disjunction: if the negation of P_1 (or of P_2, since $P_1 \vee P_2 \iff P_2 \vee P_1$) is defined, we can use the equivalence

$$P_1 \vee P_2 \iff (\neg P_1) \Rightarrow P_2$$

[10] Propositions in the assumptions must be "literals", that is, index predicates such as $a \geq 0$; they must not contain disjunctions (otherwise no theory would be convex).

to change the disjunction into an implication. However, not all constraint solvers can directly assert implications (see Section 6.6).

We are not completely pleased with this approach, as it forces users to put extraneous guards and assertions in annotations, but there may be no better way.

Disjunctive splitting is not yet implemented.[11]

6.10 The refinement restriction

In Davies' Refinement ML [Dav05a], there is a *refinement restriction* on intersection types: an intersection $A \wedge B$ is well formed only if A and B are refinements of the same simple type. For example, $\mathsf{even} \wedge \mathsf{odd}$ is permitted if even and odd both refine list; likewise, $(\mathsf{even} \to \mathsf{odd}) \wedge (\mathsf{odd} \to \mathsf{even})$ is permitted, since each component of the intersection refines $\mathsf{list} \to \mathsf{list}$. On the other hand, $\mathsf{list} \wedge (\mathsf{list} \to \mathsf{list})$ and $\mathsf{int} \wedge \mathsf{string}$ do not satisfy the refinement restriction; in the first, lists and functions are incompatible, while int and string are distinct base types. This sits well with Davies' terminology, in which the "types" are still the usual ML types, and the corresponding, more precise entities in the refinement system are called "sorts". Because of the refinement restriction, typechecking in Refinement ML is conservative over Standard ML in the following sense: every program that is well typed in Refinement ML is also well typed in Standard ML.[12] Therefore, Refinement ML programs can be processed by ordinary SML compilers. In addition, Davies' system can exploit SML's Hindley-Milner type inference for polymorphic instantiations (see Section 8.1.1), which would be difficult if the program were not a valid SML program.

In contrast, Stardust does not enforce a refinement restriction on intersections and unions. Stardust also does not check code that it knows (via rules contra, $\bot\mathbb{L}$, and the pattern rules) to be dead. Thus, it is *not* conservative in the sense that Refinement ML is—a number of programs typecheck in Stardust that do not typecheck in Standard ML, ranging from the silly

This function is in restriction1.rml.

```
(*[ val f : bool(true) → int ]*)
fun f x = case x of true ⇒ 0 | false ⇒ () ()
```

to the mildly intriguing implementation of functional arrays (as lists) shown below. The `dispatch` function is inspired by Reynolds' representation of arrays as intersections in Forsythe [Rey96], but differs in having an explicit "selector" argument to pick out the component (length, read, or write), giving it something of an object-oriented flavor.

Listing 6.2: restriction2.rml

```
(*[ datacon Nil : list(0)
    datacon Cons : -all u:nat- int * list(u) → list(u + 1)
    datatype list with nat  ]*)
datatype list = Nil | Cons of int * list

(*[ datasort array_op : length_op < array_op; read_op < array_op; write_op < array_op
    datacon Length : length_op
```

[11] Some of the machinery is present in the system, but even trivial examples requiring splitting do not work.

[12] The Standard ML compiler may emit "nonexhaustive match" warnings that the Refinement ML system knows are spurious, but it will still compile such programs.

```
    datacon Read : read_op
    datacon Write : write_op ]*)
datatype array_op = Length | Read | Write

(*[ datacon Array : -all n:nat- int(n) * list(n) → array(n)
    datatype array with int ]*)
datatype array = Array of int * list ;

(*[ val append : -all a, b : nat- list(a) * list(b) → list(a + b) ]*)
fun append (xs, ys) =
  case xs of
    Nil ⇒ ys
  | Cons(x, xs') ⇒ Cons(x, append(xs', ys))

(*[ val nth : -all len : nat- list(len) → -all n : nat- {n < len} int(n) → int ]*)
fun nth list n =
  case list of
    Cons(h, t) ⇒ if n = 0 then h else nth t (n - 1)

(*[ val replace : -all len : nat-
                  list(len) → -all n : nat- {n < len} int(n) → int → list(len) ]*)
fun replace list n x =
  case list of
    Cons(h, t) ⇒ if n = 0 then Cons(x, t) else Cons(h, replace t (n - 1) x)

(*[ val dispatch :
          -all len:nat-
            array(len)
           → ((length_op → int(len))
             & (read_op → -all n : nat- {n < len} int(n) → int)
             & (write_op → -all n : nat- {n < len} int(n) → int → array(len))) ]*)
fun dispatch array selector =
  case array of Array(length, list) ⇒
    case selector of
      Length ⇒ length
    | Read ⇒ nth list
    | Write ⇒ (fn n ⇒ fn x ⇒ Array(length, replace list n x))
```

Listing 6.2: restriction2.rml

Our next and final example might look like valid Standard ML, but does not typecheck due to that language's half-baked operator overloading.

This function is in restriction3.rml.

```
(*[ val add3 : (-all a, b, c : int- int(a) * int(b) * int(c) → int(a + b + c))
             & (-all d : dim- real(d) * real(d) * real(d) → real(d)) ]*)
fun add3 (x, y, z) = x + y + z
  (* SML arbitrarily prefers the integer variant of +, inferring
       val add3 : int * int * int → int *)

val x = add3 (10, 100, 1000)
```

```
val y = add3 (12.3, 100.0, 3.1267)   (* rejected by SML *)
```

The immediate utility of these examples is nil, since none of them can presently be compiled. However, they give some clues as to the possible benefits of building compilers based on our type system.

6.11 Performance

In this section, we give the time needed to typecheck several example programs (Figure 6.5), and discuss some of the factors affecting performance.

	Wall-clock time in seconds		
		CVC Lite	CVC Lite
Input program	ICS	(library)	(standalone)
redblack	< 1	< 1	< 1
redblack-full	1.9	8.2	9.2
redblack-full-bug1	1.6	6.8	8.1
rbdelete	*	37.7	31.6
real	< 1	< 1	< 1
refine	< 1	< 1	< 1
bits	*	9.5	4.0
bits-un	33.5	298.5	241.4

Figure 6.5: Time required for typechecking

The times indicated are under Standard ML of New Jersey version 110.59 on a 4-CPU Intel Xeon (3 GHz) and 2 GB RAM. The constraint solvers were ICS version 2.0 (November 2003) and CVC Lite version 20070121 (January 2007). An asterisk (*) indicates programs for which the constraint solver, or the interface to the constraint solver, gives a wrong answer or is deficient[13]

'redblack' implements red-black tree insertion and contains only datasort refinements (and therefore does not check the black height invariant). 'redblack-full' also contains index refinements, checking the black height invariant; this is the program in Listing 7.4. 'redblack-full-bug1' is the same program with a bug introduced. 'rbdelete' implements red-black tree deletion, using both datasort and index refinements; see Listing 7.5. 'real' contains simple functions of real numbers; it and several other (unlisted) files containing dimension type annotations do not use intersection and union types, and typechecking is consequently fast. 'refine' contains a few very short functions (with the even/odd datasort refinement of list), repeated several dozen times; each function is fast to typecheck, and the functions are checked independently, so typechecking the entire file takes very little time. 'bits' and 'bits-un' are the programs in Listings 7.1 and 7.3.

[13] The problem with ICS and `rbdelete` is caused by the use of Boolean sorts, which are not completely handled by Stardust. In contrast, the problem with bits.rml is triggered by the asserting type `[len >= len1 and len >= len2]` in the type of *add*. With the guard `[len >= len1]` alone, and with `[len >= len2]` alone, ICS works correctly. We cannot completely rule out a bug in Stardust rather than ICS, but the fact that the system behaves as expected when CVC Lite is used instead is suggestive.

6.11.1 Impact of solver interfaces

Stardust communicates with ICS through Unix pipes. This is not very efficient: experiments[14] suggest that the overhead of sending one command and receiving one response is 20–40% for ICS.

We can also communicate with CVC Lite through Unix pipes, but we have also implemented a direct interface to a shared library through CVC Lite's C-level API and the SML/NJ NLFFI. As we expected, this speeds typechecking in most cases; bits.rml is an anomaly, which might be explained by additional swapping; we manually "manage" CVC Lite expressions by never deallocating them!

For CVC Lite, another source of inefficiency is CVC Lite's inability to rapidly switch back to previously visited contexts. Unlike ICS, in which contexts are persistent and can be recalled instantaneously, CVC Lite can roll back only to ancestors of its current context. Suppose we assert $a < 0$, yielding a context Ω_1, and then assert $b < a$, yielding a state Ω_2 in which $a < 0$ and $b < a$. We can roll back to Ω_1 in a single transaction, but if we assert $c < a$, we cannot go back to Ω_2 without rolling back to Ω_1 and re-asserting (replaying) $b < a$. Typically, 20–50% of transactions with CVC Lite are replay assertions.[15] This suggests that for our purposes, persistent context in a constraint solver is useful but not absolutely essential.

Stardust also supports solvers running remotely, a feature motivated by our inability to run CVC Lite on an antiquated version of Linux; the machine in question subsequently had its OS upgraded, so the feature is disused. This was very easy to implement since communication through a Unix socket to a remote host is so similar to communication through a pipe to a local process. Typechecking using a remote solver is impractical; for example, checking `redblack` with CVC Lite running remotely on a host connected by local Ethernet takes over four minutes, due to the high cost of continually sending constraints. This does not mean that distributed typechecking in general is impractical—in fact, we outline a plausible approach with lower communication cost in Section 6.11.3—but the solver(s) should run on the same host as the Stardust process itself.

6.11.2 Conservation of speed

We believe that the system conserves typechecking speed, in the sense that checking a program—more usefully, a block—that does not use property types will be polynomial time (as with monomorphic SML programs). This is subject to the caveat that property types appear in lib_basis.rml, so any block that uses basis identifiers such as + is really using property types, whether the user realizes it or not.

The foundation of this (unproven) claim is that we believe our formal systems have a subformula property [Pra65, p. 53]: formulas (here, type expressions)—and, therefore, connectives like $\wedge$—appear in parts of a derivation only if they appear as subformulas of the goal: to show $\Gamma \vdash v \downarrow A \wedge B$ one shows $\Gamma \vdash v \downarrow A$ and $\Gamma \vdash v \downarrow B$; both A and B are subformulas of $A \wedge B$. (For this to make sense, types in annotations should be considered part of the goal.) Our system lacks

[14] We measured the slowdown resulting from a "no-op" transaction (asserting `true` and receiving a response) being executed before every normal transaction. Even that "no-op" has to be parsed, and results in some amount of legitimate constraint solver computation, but we assume that to be negligible.

[15] Specifically, 22% for `redblack` (which does not significantly use index refinements) and about 50% for `redblack-full` and `redblack-full-bug1`, in which index refinements are pervasive. These figures were arrived at with the standalone version, but should apply to the shared library case as well.

common type system machinery that breaks the subformula property (or that at least requires the property's radical reformulation); in particular, we have no non-orthogonal subtyping rules, and we do not compute greatest lower and least upper bounds of types.

6.11.3 Scaling up

Typechecking is modular, in the specific sense that each block of mutually recursive function declarations can be checked independently of each other block. For example, given a program with two mutually recursive functions *f1*,*f2* followed by a function *g*, i.e. **fun** *f1* ... **and** *f2* ... **fun** *g*, if checking *g* fails, it cannot be blamed on a choice made while checking *f1* and *f2*.

Modularity leads us to forbid contextual typing annotations on top-level declarations. Suppose we have (*[**val** $f : (\Gamma_1 \vdash A_1), (\Gamma_2 \vdash A_2)$]*) at the top level. Depending on the choice of first or second typing, we may have either $f{:}A_1$ or $f{:}A_2$ in Γ when checking subsequent blocks, which could force us to backtrack and try the other typing. In any case, there seems to be no situation in which a contextual annotation would be useful at the top level.

Thus, while property types can make checking a particular block very slow, adding a second block of the same complexity will only double typechecking time. Of course, work remains to be done to make the system fast enough to be truly usable, but this "block independence" should mean that once we have acceptable efficiency for typical programs of a few hundred lines, only linear speedup will be required to be acceptably efficient on larger programs. Moreover, we should be able to get that linear speedup through an easy form of distributed computation: If we send each block to a different processor for typechecking, the communication cost will be low, since the input is little more than the environment Γ and the block's abstract syntax, and the output is tiny: typechecking either succeeds, or fails with some error information, for that block.

There is one class of exceptions to the above: unannotated declarations of the form **val** $x = e$ where e is a synthesizing nonvalue can produce more than one type. For example, if $f : (\mathbf{1} \rightarrow B_1) \wedge (\mathbf{1} \rightarrow B_2)$ then **val** $x = f()$ will yield first $x{:}B_1$ and—if subsequent typechecking fails—$x{:}B_2$. This would have to be taken into consideration in a parallelization scheme. Alternatively, we could require annotations on *all* top-level declarations, even synthesizing nonvalues such as the above. With a module system, exported top-level declarations need annotations in the module signature anyway, so this is not as extreme as it might appear. Another option would be to require an annotation only if the bound expression has no principal type; if we had $f : \mathbf{1} \rightarrow B$ instead, the bound expression $f()$ would principally synthesize B. Of course, this option assumes we can easily determine if a bound expression has a principal type.

6.12 Error reporting

Error reporting is currently very limited. When a program does not pass the typechecker, the system gives only the locations (range of line numbers) of the ill-typed block and the last expression examined. Two factors make good reporting difficult: index constraints and intersection/union types.

Index constraints contain index variables generated internally, making them hard to decipher. However, one can maintain a map from index variables to the program locations at which they

were generated. For example, if we apply a function $f : \Pi a{:}\gamma.\, A \to B$ in the term $f\ x$, the existential variable created for a should map to the line containing $f\ x$. Then, when typechecking fails due to a constraint P being invalid, we can report the locations associated with the free index variables in P. This seems to work fairly well in Xi's DML [Xi98].

Intersections and unions cause multiple obligations. Checking against an intersection $A_1 \wedge A_2$ requires checking each part; one would like to know which part failed. That is straightforward. However, in checking against a union $B_1 \vee B_2$ the typechecker tries to check against B_1 and, if that fails, B_2. We report an error when both fail; simply reporting the location of the last error would ignore B_1. If unions and intersections alternate in the type checked against, as with $(A_1 \wedge A_2) \vee (C_1 \wedge C_2)$, we should report appropriately: "In $A_1 \wedge A_2$, could not check against A_2 because . . . ; in $C_1 \wedge C_2$, could not check against C_1 because . . . ", with the right program locations for each part. Such alternations make reporting harder than in Davies' Refinement ML, which has intersections only [Dav05a].

Even with these issues, bidirectionality gives us some advantage over typecheckers based on unification. As Davies [Dav05a] and Pierce and Turner [PT98] have observed, in a bidirectional system, the location reported is more likely to be the real cause of the error, rather than hundreds of lines away as is sometimes the case with unification.

6.13 Parametric polymorphism

Like the type systems in previous chapters, Stardust does not support parametric polymorphism. No type quantifiers or type schemes can appear in annotations. Such things are not fabricated, satisfying the subformula property. Our preliminary ideas for how to add parametric polymorphism to the system likewise do not involve producing polymorphism out of thin air; see Section 8.1.1.

Acknowledgment

Portions of Stardust are based on code from Tolmach and Oliva's "Restricted ML" compiler [TO98], known as RML (hence the `.rml` filename suffix), which should not be confused with Davies' "Refinement ML" [Dav05a].

Chapter 7

Index domains

7.1 Introduction

Our type system, like Xi's DML [Xi98], is parametric in the index domain. Our implementation (Chapter 6) supports two major index domains: integers and dimensions. The integer domain (Section 7.2) was studied and implemented in DML, but the application of index refinements to dimension types, discussed in Section 7.4, is novel. We also support a domain of Booleans (Section 7.3).

Each section includes examples typed with refinements from the relevant domain. Of these, bitstrings (Section 7.2.3) and red-black tree insertion and deletion (Sections 7.2.4 and 7.2.5) may be of particular interest, and those sections can reasonably be read independently of the surrounding material.

7.2 Integers

We write $\mathcal{Z}$ for the index sort of integers, which has the following structure:

	mathematical notation	Stardust notation
Index sort	$\mathcal{Z}$	`int`
Constants	$\ldots, -2, -1, 0, 1, 2, \ldots$	`...,~2,~1,0,1,2,...`
Functions	$+,\ -,\ *,\ /$	`+`, `-`, `*`, `/`
Predicates	$<,\ \leq,\ \doteq,\ \neq,\ \geq,\ >$	`<`, `<=`, `=`, `<>`, `>=`, `>`

Nonlinear expressions such as `a * b` must be avoided to ensure that the resulting constraints are decidable.

The integer sort refines the base type `int`. This is something of a mismatch, since `int` is a type of *finite* integers. The more direct application of $\mathcal{Z}$ would be to index `IntInf`, SML's arbitrary-precision integer type. We are saved from a truly worrisome situation by the fact that SML integer arithmetic operations must raise an exception on overflow [GR04, pp. 170–171], so the results of such operations will have the values their integer indices claim they do; if $x = 5$ then $x + 1 = 6$ as claimed, while if $x =$ `maxInt` then $x + 1$ raises an exception—no harm done. The only major

shortcoming is that the type system cannot prove that overflow will not happen; this is not trivial, because there is presently no way in our system to assert anything about the exception-raising properties of terms. Likewise, in the body of **handle** Overflow $\Rightarrow \ldots$ the type system does not know that overflow occurred.

Bit vectors might be a better index domain for `int`; even if we have to encode every integer index variable as 32–64 Boolean variables, addition and subtraction should be no trouble for SAT solvers designed to handle enormous formulas. As with the current state of affairs, it does not matter if overflow occurs in the ML program as an exception will be raised (in fact, the constraint solver could produce nonsense in overflow situations without making the system unsound). However, bit vectors would be perfect for types in which arithmetic operations are supposed to silently "roll over" without raising an exception, such as `word` in SML and `int` in OCaml.

7.2.1 Natural numbers

The definition is in lib_basis.rml.

A sort of natural numbers, $\mathcal{N}$, is defined as the *subset sort* $\{a{:}\mathcal{Z} \mid a \geq 0\}$:

```
(*[
  ...
  indexsort nat = {a:int | a >= 0}
]*)
```

As described in Section 6.3.4, the Stardust system removes subset sorts by transforming type expressions, adding guarded and asserting types. Thus, when an index variable a of sort $\mathcal{N}$ is declared to the underlying constraint solver, the proposition $a \geq 0$ will be either assumed or checked (for example, when checking the body of a function whose argument or result, respectively, has type $\delta(a)$). Other subsets, such as the strictly positive numbers, can be defined in the same way.

7.2.2 Implementation

In Stardust, most of the work of integer constraint solving is handled by external solvers (ICS or CVC Lite); we discussed Stardust's interface to external solvers, and issues specific to integers, in Section 6.6. We add only that this is a well-studied area; see, for instance, Dantzig and Eaves [DE73] and Pugh [Pug91].

7.2.3 Example: Inductive bitstrings

Adapting an example of Davies et al. [DP00, Dav05a], we show how datasort and index refinements in combination can elegantly capture data structure invariants. In this example, E represents the empty bitstring and Zero, One add to the end of the string: for example, One(E) represents the bitstring 1 or `e1` (rendering the "empty part" of the string as `e`), and Zero(One(E)) represents `e10`.

A datasort refinement distinguishes bitstrings without leading zeroes (std, for "standard form"), as well as those that have no leading zeroes and are also not the empty bitstring (pos). Index refinements encode both the bitstring's length and its value, as a natural number. For example, One(One(E)), which is two bits long and represents 3, has type pos$(2, 3)$.

The example, bits.rml, is shown in Listing 7.1. We can give refined types for several functions on bitstrings:

- For the type of the increment function inc, we need to give the datasort, length, and value refinements of its output. The datasort and value are straightforward enough (since inc is supposed to increment the bitstring by one, the value represented by the result is the value of the input + 1); for the length, which might or might not increase by one, we use union types.[1]

- For the addition function add we write a rather vague type; a more accurate type, using unions, is given in bits-un.rml (Listing 7.3), but the type of inc is insufficient to check add against the more accurate type. The sticking point is in the last arm of add:

  ```
  | (x, One y') ⇒ inc (add (x, Zero y'))
  ```

  We want to show, given x of length len1 and y' of length len2 - 1, that $\texttt{inc}(\texttt{add}(x, \mathsf{Zero}\ y'))$ has length $\leq max(\texttt{len1}, \texttt{len2}) + 1$. The difficulty is that, in general, add may return a bitstring longer than both its arguments, and inc may return a bitstring longer than its argument. However, incrementing the result of a "lengthening addition" cannot again lengthen the result:

  - If $x = y$ then $\texttt{add}(x, y)$ will be even, i.e. $\mathsf{Zero}(z)$; but then $\texttt{inc}(\mathsf{Zero}(z)) = \mathsf{One}(z)$, which is no longer than $\mathsf{Zero}(z)$.
  - On the other hand, if $x \neq y$, suppose w.l.o.g. that $x > y$. Then $\texttt{len1} \geq \texttt{len2}$. We have $y < x < 2^{\texttt{len1}}$. Because y is strictly less than x, the sum of x and y is strictly less than twice x, so $x + y < 2^{\texttt{len1}+1} - 1$. Thus the length of $x + y$ is at most $\texttt{len1} + 1$. Moreover, *if* the length is $\texttt{len1} + 1$, the bitstring $x + y$ must have at least one zero bit, because the only bitstring of length $\texttt{len1}+1$ that does not have any zeroes has value $2^{\texttt{len1}+1} - 1$—and we showed that $x + y$ is strictly less than that. Therefore:
    - If $\texttt{add}(x, y)$ returns a bitstring of length $\texttt{len1} + 1$, it will have a zero. When inc is applied to an argument of length $\texttt{len1} + 1$ with a zero, its result also has length $\texttt{len1} + 1$.
    - If $\texttt{add}(x, y)$ returns a bitstring of length len1, it may or may not have a zero; inc will return a bitstring of length len1 or $\texttt{len1} + 1$.

    In both cases, the length of $\texttt{inc}(\texttt{add}(x, y))$ must be len1 or $\texttt{len1} + 1$.

  It might be helpful to capture, with a datasort refinement, the property of a bitstring being all ones or having at least one zero. This is relevant because inc's result is longer than its argument iff its argument is all ones.

  A more clearly promising alternative would be to use an index domain including logarithms and powers of 2. Then we can remove the length entirely (provided we force all bitstrings to

[1] As an aside, it is clear that the type of inc contains redundant information. We are really specifying three distinct properties: (1) that inc takes std to pos and bits to bits; (2) that the length of its result is either the same or one more than its argument; (3) that the value of its result is one plus the value of its argument. We leave a more concise means of writing types that thus "orthogonally" combine properties as future work.

be in standard form std), since it is always given by $\lceil \log_2(\texttt{value} + 1) \rceil$. This is played out in Listing 7.2, which of course does not fully typecheck because we lack a constraint solver for such a domain.

- The functions `toInt` and `length`, which convert a bitstring to an integer and obtain its length, are easy to type, using singletons in the ranges.
- The function to produce a bitstring *from* a nonnegative integer is problematic in our index domain. We can easily write a type expressing that the result is in standard form (i.e. no leading zeroes) and represents the given integer. We leave the length unspecified since that would need a logarithm operation ($\lceil \log_2(\texttt{value} + 1) \rceil$). Unfortunately, checking the datasort and value-index refinements requires division and remainder index operations. If we had those operations, we would be able to check the `fromInt` function in Listing 7.2.

Listing 7.1: bits.rml

```
(* bits.rml –– Inductively defined bitstrings, with:

    (1) a datasort refinement:
        bits = any bitstring,
        std = "standardized" bitstrings, i.e. with no leading zeros,
        pos = standardized bitstrings of nonzero value (i.e. not the empty bitstring).

    (2) index refinement: (length, value) where
        length : nat = the length of the bitstring,
        value : nat = the number represented. *)
(*[
  datasort  bits : pos <= std; std <= bits
  datacon  E : std(0, 0)
  datacon  Zero : -all  len, value : nat- pos(len, value) → pos(len+1, 2*value)
                                   &  bits(len, value) → bits(len+1, 2*value)
  datacon  One : -all  len, value : nat- std(len, value)  → pos(len+1, 2*value + 1)
                                  &  bits(len, value) → bits(len+1, 2*value + 1)
  datatype  bits with  nat * nat
 ]*)
datatype  bits =
   E
 | Zero of  bits
 | One of  bits
;

(*[ val  xx : std(0, 0) ]*)    val  xx = E
(*[ val  yy : bits(2, 1) ]*)   val  yy = One (Zero (E))  (* e01 : Leading zero *)
(*[ val  zz : pos (2, 2) ]*)   val  zz = Zero (One (E))  (* e10 : No leading zero *)
(*[ val  zz' : pos(6, 43) ]*)  val  zz' = One (One (Zero (One (Zero (One (E))))))
         (* e101011 : No leading zero *)

(*[ val  inc : -all  len, value : nat-
                std(len, value)  → pos(len, value+1) \/ pos(len+1, value+1)
             &  bits(len, value) → bits(len, value+1) \/ bits(len+1, value+1)
]*)
fun  inc n =
```

```
  case n of
    E => One E
  | Zero n => One n
  | One n => Zero (inc n)

(*[ val add : -all len1, len2, val1, val2 : nat-
                    bits(len1, val1) * bits(len2, val2)
              -> (-exists len : nat- [len >= len1 and len >= len2]
                    bits(len, val1 + val2))
]*)
fun add arg = case arg of
      (x, E) => x
    | (E, y) => y
    | (Zero x', Zero y') => Zero (add (x', y'))
    | (One x', Zero y') => One (add (x', y'))
    | (x, One y') => inc (add (x, Zero y'))

(*[ val toInt :! -all len, value : nat- bits(len, value) -> int(value)
]*)
fun toInt n = case n of
     E => 0
   | Zero n' => 2 * toInt n'
   | One n' => (2 * toInt n) + 1  (* BUG *)

(*[ val toInt : -all len, value : nat- bits(len, value) -> int(value)
]*)
fun toInt n = case n of
     E => 0
   | Zero n' => 2 * toInt n'
   | One n' => (2 * toInt n') + 1

(*[ val length : -all len, value : nat- bits(len, value) -> int(len)
]*)
fun length n = case n of
     E => 0
   | Zero n' => 1 + length n'
   | One n' => 1 + length n'

(* Can't properly check spec because
     div, mod not supported by constraint solver

   (*[ val fromInt : -all value : nat- int(value)
                                    -> -exists len : nat- std(len, value)
   ]*)
*)
(* Weak specification: *)
(*[ val fromInt : int -> bits
]*)
fun fromInt n =
  if n = 0 then E
  else
      let val d = fromInt (n div 2)
          val r = n mod 2
      in
```

```
        if  r = 0 then  Zero d
        else  One d
    end
```

Listing 7.1: bits.rml

Listing 7.2: unconstrained-bits.rml

```
(* unconstrained—bits.rml —— Inductively defined bitstrings, with:

   (1) a datasort refinement:
       bits = any bitstring,
       std = "standardized" bitstrings, i.e. with no leading zeros,
       pos = standardized bitstrings of nonzero value (i.e. not the empty bitstring).

   (2) index refinement: (value) where
       value : nat = the number represented.

   This is a "dead" example: it assumes an index domain that
   includes div, mod, and constant—base exponentiation (2**n), logarithms (log_2(n)),
   and ceiling ceil(n), and we do not have a constraint solver for that domain.

   The functions 'inc', 'add', and 'toInt' do not use those operations, so we
   *can* check their types.
*)
(*[
  datasort  bits : pos <= std; std <= bits
  datacon  E : std(0)
  datacon  Zero : -all  value : nat- pos(value) → pos(2*value)
                                   &  bits(value) → bits(2*value)
  datacon  One : -all  value : nat- std(value)  → pos(2*value + 1)
                                  &  bits(value) → bits(2*value + 1)
  datatype  bits with  nat
 ]*)
datatype  bits = E | Zero of bits | One of bits ;

(*[ val xx : std(0) ]*)    val  xx = E
(*[ val yy : bits(1) ]*)   val  yy = One (Zero (E))  (* e01 : Leading zero *)
(*[ val zz : pos (2) ]*)   val  zz = Zero (One (E))  (* e10 : No leading zero *)
(*[ val zz' : pos(43) ]*)  val  zz' = One (One (Zero (One (Zero (One (E))))))
         (* e101011 : No leading zero *)

(*[ val inc : -all  value : nat-
                std(value)  → pos(value+1) \/ pos(value+1)
              & bits(value) → bits(value+1) \/ bits(value+1)
]*)
fun inc n =
  case  n of
    E ⇒ One E
  | Zero n ⇒ One n
  | One n ⇒ Zero (inc n)

(*[ val add : -all  val1, val2 : nat-
                  bits(val1) * bits(val2)  →  bits(val1 + val2)
]*)
```

```
fun add arg = case arg of
      (x, E) => x
    | (E, y) => y
    | (Zero x', Zero y') => Zero (add (x', y'))
    | (One x', Zero y') => One (add (x', y'))
    | (x, One y') => inc (add (x, Zero y'))

(*[ val toInt :! -all value : nat- bits(value) -> int(value)
]*)
fun toInt n = case n of
     E => 0
   | Zero n' => 2 * toInt n'
   | One n' => (2 * toInt n) + 1  (* BUG *)

(*[ val toInt : -all value : nat- bits(value) -> int(value)
]*)
fun toInt n = case n of
     E => 0
   | Zero n' => 2 * toInt n'
   | One n' => (2 * toInt n') + 1

(* Everything above this line does not use log_2 / 2 ** / div / mod,
 and can be typechecked by Stardust. *)

(*[ val length : -all value : nat- std(value) -> int(ceil(log_2(value + 1)))
]*)
fun length n = case n of
     E => 0  (* ceil(log_2(value + 1)) = ceil(log_2(0 + 1)) = ceil(0) = 0 *)
   | Zero n' => 1 + length n'
   | One n' => 1 + length n'

(*[ val fromInt : -all value : nat- int(value) -> std(value)
]*)
fun fromInt n =
  if n = 0 then E
  else
      let val d = fromInt (n div 2)  (* d : std(value div 2) *)
          val r = n mod 2  (* r : -exists a : nat- [a < 2] int(r) *)
      in
          if r = 0 then Zero d
          else One d
      end
```

Listing 7.2: unconstrained-bits.rml

Listing 7.3: bits-un.rml

```
(* bits-un.rml

   Ill-typed 'add' function (see bits.rml) with union types.
*)

(*[
  datasort bits : pos <= std; std <= bits
  datacon E : std(0)
```

```
  datacon  Zero : -all  len : nat- pos(len) → pos(len+1)
                               &  bits(len) → bits(len+1)
  datacon  One : -all  len : nat- std(len)  → pos(len+1)
                               &  bits(len) → bits(len+1)
  datatype  bits with  nat
 ]*)
datatype  bits =
   E
 | Zero of bits
 | One of bits
;

(*[ val inc : -all  len : nat-
               std(len)  → pos(len) \/ pos(len+1)
            &  bits(len) → bits(len) \/ bits(len+1)
]*)
fun inc n =
  case  n of
    E ⇒ One E
  | Zero n ⇒ One n
  | One n ⇒ Zero (inc n)

(*[ val add : -all  len1, len2 : nat-
                  bits(len1) * bits(len2)
             →
                  bits(len1)
               \/ bits(len1 + 1)
               \/ bits(len2)
               \/ bits(len2 + 1)
]*)
fun add arg = case  arg of
      (x, E) ⇒ x
    | (E, y) ⇒ y
    | (Zero x', Zero y') ⇒ Zero (add (x', y'))
    | (One x', Zero y') ⇒ One (add (x', y'))
    | (x, One y') ⇒  inc (add (x, Zero y'))
```

Listing 7.3: bits-un.rml

7.2.4 Example: Red-black tree insertion

Red-black trees are another good example of datasort and index refinements in combination. We will use the datatype declaration that follows. We fix the key type to be int, and we have no associated record component, so that a dict represents a set of integers rather than a map from integers to some other type. Since our refinements will be concerned with the *structure* of the trees rather than the integers contained—we will not try to guarantee an order invariant, for example—having a set rather than a map is not detrimental to the example.

```
datatype dict =
    Empty
```

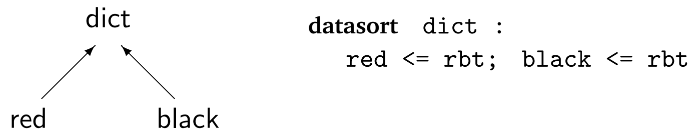
```
  | Black of int * dict * dict
  | Red of int * dict * dict
```

We use refinements to guarantee the second and third invariants given in Listing 7.4:

2. The children of a red node are black (color invariant).
3. Every path from the root to a leaf has the same number of black nodes, called the black height of the tree.

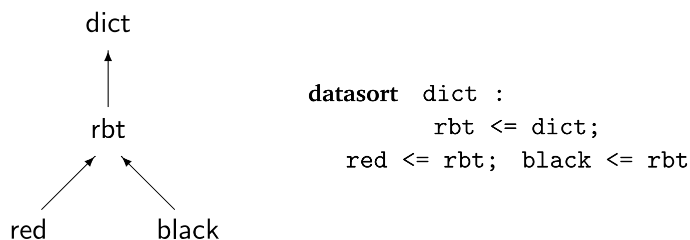
Every tree satisfying these invariants is balanced: the height of a tree containing n non-Empty nodes is at most $2\log_2(n+1)$ [GS78, CLR90, p. 264]. Invariant 2 is concerned with color, and the colors of a datasort form a small finite set, so it is a suitable candidate for a datasort refinement. Invariant 3 involves node color, but also black height, which is a natural number and therefore suitable for index refinement.

The obvious datasort refinement might be



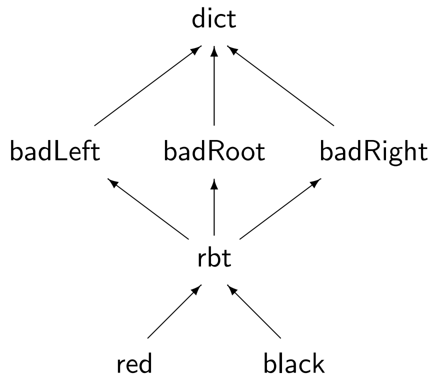

```
datasort dict :
   red <= rbt;  black <= rbt
```

With the appropriate types for Red and Black we could ensure that only trees satisfying the invariants are constructed. However, it will be useful to create, on a temporary basis, trees that do not satisfy the color invariant (2). (Invariant 3, about black height, is still guaranteed for all trees.) So we add a datasort rbt of "proper" **r**ed-**b**lack **t**rees, which satisfy invariants 2 and 3. In the following datasort declaration, red and black are subsorts of rbt, so they (continue to) represent proper red-black trees of known color.




```
datasort dict :
          rbt <= dict;
   red <= rbt;  black <= rbt
```

However, we need one further refinement of the datasorts. The relation above distinguishes trees that satisfy all the invariants (rbt) from those that might not satisfy the color invariant (dict); if we know something is a dict we know nothing about *where* the color violation occurs. Thus, even if *we* know that a color violation, if any, in the argument x of a function f : dict $\rightarrow$ rbt, if any, occurs at the root of x, Stardust cannot—and even if f deals properly with such violations at x's root, as far as Stardust is concerned there may still be violations elsewhere in x. So we add datasorts badRoot, badLeft and badRight for possible color violations at the root (the root is red and some child is red), at the left child (the left child is red and one of *its* children is red), and at the right child (*mutatis mutandis*). Note that the "good" datasorts rbt, red, black are subsorts of the "bad" datasorts badRoot, etc.: the "bad" datasorts represent not that the color invariant is violated, but that it *may* be violated.




```
datasort dict :
  badLeft <= dict; badRoot <= dict; badRight <= dict;
   rbt <= badLeft; rbt <= badRoot; rbt <= badRight;
            red <= rbt;  black <= rbt
```

Listing 7.4: redblack-full.rml

```
(* redblack−full.rml

   Based on an example of Rowan Davies and Frank Pfenning
*)
(*[ datatype dict with nat

    datasort dict :
          badLeft <= dict; badRoot <= dict; badRight <= dict;
           rbt <= badLeft;  rbt <= badRoot; rbt <= badRight;
                      red <= rbt;   black <= rbt

    datacon Empty : black(0)

    datacon Black :
            -all h : nat- int * dict(h) * dict(h) → dict(h+1)
                       &  int * rbt(h) * rbt(h) → black(h+1)
                       &  int * badRoot(h) * rbt(h) → badLeft(h+1)
                       &  int * rbt(h) * badRoot(h) → badRight(h+1)

    datacon Red :
            -all h : nat- int * dict(h) * dict(h) → dict(h)
                       &  int * black(h) * black(h) → red(h)
                       &  int * rbt(h) * black(h) → badRoot(h)
                       &  int * black(h) * rbt(h) → badRoot(h)
]*)
datatype dict =
    Empty
  | Black of int * dict * dict
  | Red of int * dict * dict
;
  (* Representation Invariants

      1. The tree is ordered: for every node (Red|Black)(key1, left, right),
         every key in left is less than key1 and
         every key in right is greater than key1.

      2. The children of a red node are black (color invariant).

      3. Every path from the root to a leaf has the same number of
         black nodes, called the black height of the tree.
```

```
*)

  (*[ val  lookup : rbt → int → bool ]*)
  fun lookup dict key =
    let
      (*[ val lk : rbt → bool
        val lk' : int * rbt * rbt → bool ]*)
      fun lk dict =
          case dict of
              Empty ⇒ false
            | Red tree ⇒ lk' tree
            | Black tree ⇒ lk' tree
      and lk' (key1, left, right) =
          if key = key1 then true
          else if key < key1 then lk left
          else lk right
    in
        lk dict
    end

(*[ val restore_right : -all h : nat- badRight(h) → rbt(h)
]*)
  (* restore_right (Black(e,l,r)) ⟹ dict
      where (1) Black(e,l,r) is ordered,
            (2) Black(e,l,r) has black height n,
            (3) color invariant may be violated at the root of r:
                one of its children might be red.
      and dict is a re−balanced red/black tree (satisfying all invariants)
      and same black height n. *)

  fun restore_right arg = case arg of
       Black(e, Red lt, Red (rt as (_,Red _,_))) ⇒
         Red(e, Black lt, Black rt)  (* re−color *)

(* EXAMPLE BUG: Black lt instead, above. Not caught: black height same as correct version. *)
(* EXAMPLE BUG: Empty instead, above. Caught. *)

    | Black(e, Red lt, Red (rt as (_,_,Red _))) ⇒
        Red(e, Black lt, Black rt)  (* re−color *)

    | Black(e, l, Red(re, Red(rle, rll, rlr), rr)) ⇒
         (* l is black, deep rotate *)
         Black(rle, Red(e, l, rll), Red(re, rlr, rr))

    | Black(e, l, Red(re, rl, rr as Red _)) ⇒
         (* l is black, shallow rotate *)
         Black(re, Red(e, l, rl), rr)

    | dict ⇒ dict

(* restore_left is like restore_right, except
      the color invariant may be violated only at the root of left child *)
(*[ val restore_left : -all h : nat- badLeft(h) → rbt(h) ]*)
  fun restore_left arg = case arg of
      (Black(e, Red (lt as (_,Red _,_)), Red rt)) ⇒
```

```
        Red(e, Black lt, Black rt)  (* re−color *)
  | (Black(e, Red (lt as (_,_,Red _)), Red rt)) ⇒
        Red(e, Black lt, Black rt)  (* re−color *)
  | (Black(e, Red(le, ll as Red _, lr), r)) ⇒
        (* r is black, shallow rotate *)
        Black(le, ll, Red(e, lr, r))
  | (Black(e, Red(le, ll, Red(lre, lrl, lrr)), r)) ⇒
        (* r is black, deep rotate *)
        Black(lre, Red(le, ll, lrl), Red(e, lrr, r))
 | dict ⇒ dict

(*[ val insert : rbt * int → rbt ]*)
fun insert (dict, key) =
  let
    (* val ins1 : dict → dict inserts entry *)
    (* ins1 (Red _) may violate color invariant at root *)
    (* ins1 (Black _) or ins (Empty) will be valid red/black tree *)
    (* ins1 preserves black height *)
    (*[ val ins1 : -all h : nat- rbt(h) → badRoot(h)
                                & black(h) → rbt(h)
    ]*)  (* the second conjunct is needed for the recursive cases *)
    fun ins1 arg = case arg of
        Empty ⇒ Red(key, Empty, Empty)
      | Black(key1, left, right) ⇒
          if key = key1 then Black(key, left, right)
               (* EXAMPLE BUG: change 'Black(...)' to 'left' (see redblack−full−bug1.rml *)

          else if key < key1 then restore_left (Black(key1, ins1 left, right))
          else restore_right (Black(key1, left, ins1 right))
      | Red(key1, left, right) ⇒
          if key = key1 then Red(key, left, right)
          else if key < key1 then Red(key1, ins1 left, right)
          else Red(key1, left, ins1 right)
  in
    case ins1 dict
      of Red (t as (_, Red _, _)) ⇒ Black t  (* re−color *)
       | Red (t as (_, _, Red _)) ⇒ Black t  (* re−color *)
       | dict ⇒ dict  (* depend on sequential matching *)
  end
```

Listing 7.4: redblack-full.rml

Related work

Our combination of refinements for red-black tree insertion is new, but the application of datasort and index refinements individually is not. In fact, Listing 7.4 is based on code from Davies' thesis [Dav05a, pp. 277–279]; our contribution is that the black height is also guaranteed. However, that refinement is not new either: it can be found in Xi's thesis [Xi98, pp. 161–165]. Xi also guarantees the color invariant, but by refining the tree datatype by the index product $\mathcal{Z} * \mathcal{Z} * \mathcal{Z}$, representing the color, black height, and "red height" respectively. 0 in the first component means

red and 1 means black, with an existential quantifier used if the color is not known. The so-called "red height" is not analogous to the black height, but instead is supposed to count the *consecutive* red nodes, and is to be 0 if there are none, i.e. if the color invariant is satisfied. The resulting types are awkward and substantially less legible.[2]

Type-level programming techniques not involving type refinements have also been applied to check red-black tree invariants, relying on phantom and existential types [Kah01]. At least to our untrained eyes, such approaches are more complicated and awkward than even Xi's color-encoding. Moreover, they require substantial changes to the basic tree datatype and associated code.

7.2.5 Example: Red-black tree deletion

As just discussed, others have previously studied how to refine red-black tree insertion. Deletion is another matter. Though more complicated than insertion, deletion in an imperative style can be cooked up from pseudocode in standard sources, e.g. Cormen et al. [CLR90]; purely functional deletion cannot (unlike insertion, it is not treated by Okasaki [Oka98]). However, there are implementations, such as the *RedBlackMapFn* structure in the Standard ML of New Jersey library[3]. We easily translated *RedBlackMapFn* into the Stardust subset of ML. Finding appropriate refinements and invariants was substantially more difficult; however, we eventually succeeded, resulting in an implementation of deletion that is known to satisfy invariants (1)–(3) (see the previous section on red-black tree insertion). In the course of this implementation, we found several lacunae in the New Jersey library (including two bugs leading to a violated color invariant), which we discuss at the end of this section.

A key data structure in deletion is the *zipper*, which represents a tree with a hole, *backwards*, allowing easy traversal toward the tree's root. This corresponds to the series of rotations and color changes that may be required after deleting a node. The function zip takes a zipper and a tree and plugs the tree into the zipper's hole.

```
datatype zipper
  = TOP
  | LEFTB of int * dict * zipper
  | LEFTR of int * dict * zipper
  | RIGHTB of dict * int * zipper
  | RIGHTR of dict * int * zipper
```

The TOP constructor represents a zipper consisting of just a hole; the LEFTB and LEFTR constructors represent the edge from a left child to a Black or Red node, respectively. The RIGHTB and RIGHTR constructors are symmetric. The examples in Figure 7.1 should clarify the constructors' meaning.

We refine zippers by a datasort and two integer indices.

[2] In fact Xi uses a *four*-tuple of integers, with the fourth component representing size (number of non-leaf nodes), but that would not be a fair comparison. Note that we could easily incorporate size into our refinement, leading to dict being refined by a datasort and two integers.

[3] The SML/NJ library is not part of the *standard* Basis Library, but is included with major SML compilers such as SML/NJ and MLton; its use is widespread.

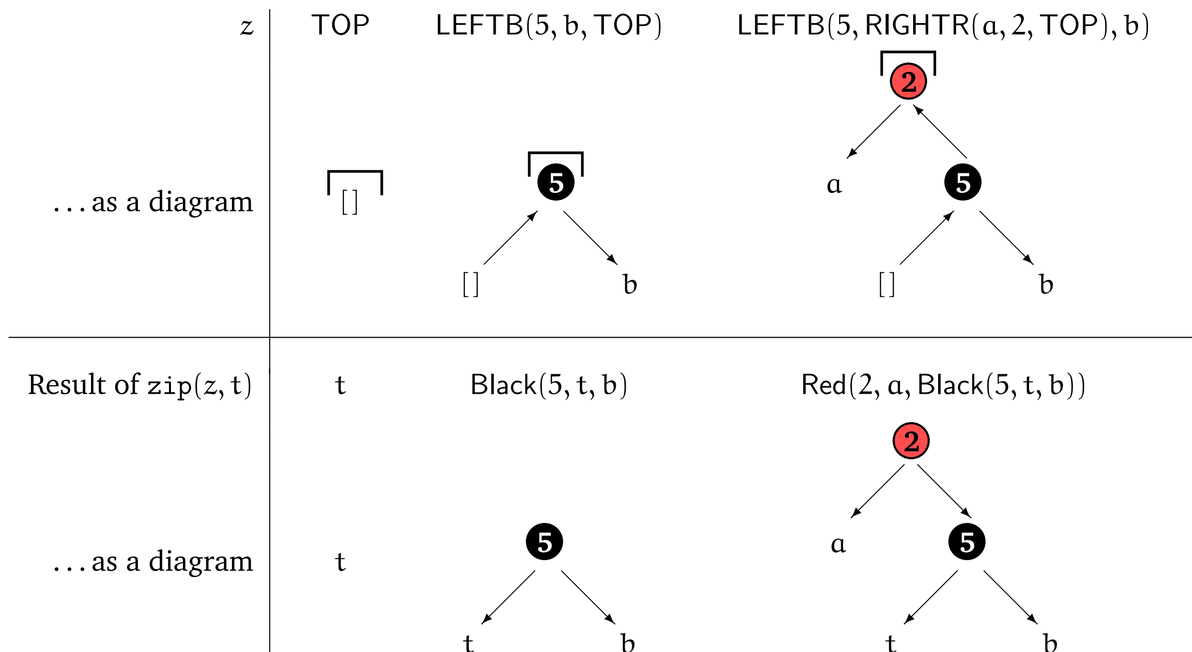


Figure 7.1: Examples of zippers

Datasort refinement of zipper

The datasort encodes two properties: the color of the hole's parent and the color of the root of the result of $\mathtt{zip}(z, t)$. If a zipper has datasort blackZipper, the root (of the zipper, that is, the parent of the hole) is black and can therefore tolerate black trees as well as red; thus, all values LEFTB(. . .) and RIGHTB(. . .) have datasort blackZipper. If a zipper z has datasort BRzipper, meaning "Black Root zipper", the resulting zipped tree $\mathtt{zip}(z, t)$ will have a black root and thus have datasort black; all zippers having the form . . . LEFTB(_, _, TOP) . . . or the form . . . RIGHTB(_, _, TOP) . . .) have datasort BRzipper.

We also have a datasort topZipper, such that TOP : topZipper. Zipping TOP with a tree t yields t itself; therefore, zipping TOP with a black tree yields a black tree. This property of a zipper yielding a black tree when it is zipped with a black tree is captured by topZipper.

The remaining datasorts, topOrBR and blackBRzipper, can be thought of as the union and intersection, respectively, of BRzipper and topZipper, as shown in Figure 7.2.

Index refinement of zipper

We refine the zipper datatype by a pair of natural numbers. A zipper z has index refinement (h, hz) precisely if zip, when given z and a tree t of black height h (that is, $t : \mathsf{rbt}(h)$), yields a tree of black height hz. Hence, TOP has type $\mathsf{zipper}(h, h)$ for all h, because $\mathtt{zip}(\mathsf{TOP}, t)$ yields just t.

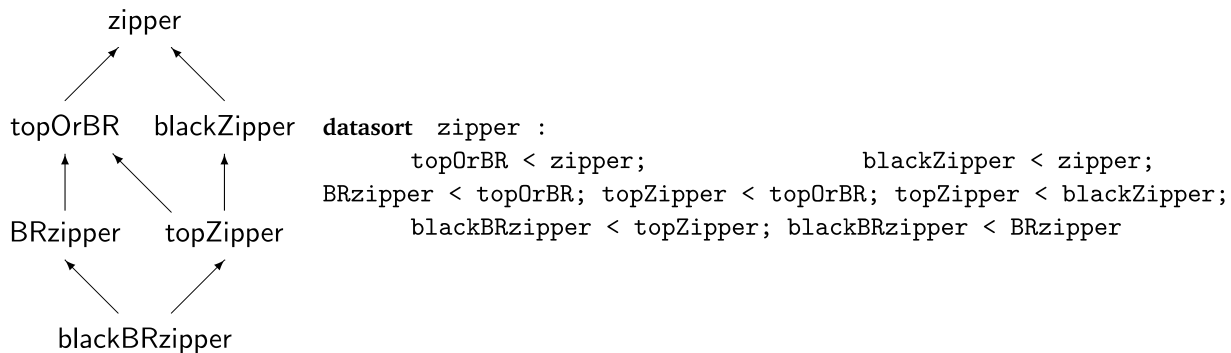


Figure 7.2: Datasort relation for zippers

Overview of the algorithm

Our code is closely based on the New Jersey library, which claimed to implement, in a functional style, the imperative pseudocode given in Cormen et al. [CLR90]. At a very high level, without regard for the red-black tree invariants, deletion of key k in the tree t goes as follows:

1. Starting from the root of t, find a node with key k.
2. Join the left and right children a, b of the node containing key k:

   i. Find the minimum x of b; this x is greater than all keys in a, and less than all other keys in b.

   ii. Delete the node containing x.

   iii. Replace the node containing k with one containing x, with the same a and the new b (with x deleted) as its children.

We can distinguish cases of deletion based on the color of the node containing x, the minimum key in the subtree rooted at b.[4] First note that since x is the minimum, the node containing it must have an empty left child.

- If the node containing x is red, its right child cannot be Red (color invariant), nor can it be Black (its left child is empty—black height 0—so its right child must also have black height 0, but any Black-rooted tree has black height at least 1), so the right child must also be empty.

  Deleting a red node does not change any black heights, so the black height invariant is preserved; the only change needed is to substitute x for k.

- If the node containing x is black, its right child cannot be Black (by similar reasoning to the red case). However, its right child could be $\mathsf{Red}(y, \mathsf{Empty}, \mathsf{Empty})$, in which case we can just replace x with y—keeping the node containing x black—which preserves black height. The hard case is when both children of the node containing x are empty: merely deleting that

[4] These cases do *not* correspond to the `joinRed` and `joinBlack` functions, whose names refer to the color of the node containing k, the key searched for, not x, the minimum element in k's right subtree.

node means that its parent will have a left subtree of black height 0 and a right subtree of black height 1, which is inconsistent. This is called a *black deficit*. We can try to fix it by calling bbZip, which moves up the tree towards the root, performing rotations and color changes. While this process will always yield a valid subtree that satisfies the black height invariant (and of course the color invariant), it may not actually "fix the deficit": the resulting subtree may still have a black height that is one less than before. If that occurs—signalled by bbZip returning (true, t)—we call bbZip again, continuing the rotations and color changes upward past the node that used to contain k (and now contains x). Otherwise, all the invariants have been fixed, and we need only replace k with x and call zip.

The zip function

The zip function is quite straightforward, but its refined type may need some explanation.

```
(*[ val zip :
      -all h, hz : nat-
          blackZipper(h, hz) * rbt(h) → rbt(hz)
        & zipper(h, hz) * black(h) → rbt(hz)
        & blackBRzipper(h, hz) * rbt(h) → black(hz)
        & topOrBR(h, hz) * black(h) → black(hz)
 ]*)
```

The index refinement is almost painfully obvious: a zipper that yields a tree of black height hz when zipped with a tree of black height h, when zipped with such a tree, yields a tree of black height hz. This is because we refined the zipper datatype with the behavior of zip in mind.

The datasorts are, perhaps, less painfully obvious. The first part of the intersection expresses the fact that if the parent of the zipper's hole is black (blackZipper) then replacing the hole with any valid tree (rbt) yields a valid tree. The second part says that if the parent of the hole is *not* known to be black, then only a black tree can be substituted, because the parent might be red and we cannot allow a color violation. The third part of the intersection says that, when a blackBRzipper—a zipper with a black node as the parent of the hole and that, when zipped, yields a black-rooted tree—is zipped with any tree, a black-rooted tree results. The fourth says that when either a topZipper (such as TOP) or a BRzipper (such as RIGHTB(a, 2, TOP)) is zipped with a black tree t, a black tree results—if the zipper is TOP, because the result consists of just t, which is black; if the zipper is BRzipper, because the result has a black root regardless of the color of t.

The bbZip function

bbZip is a recursive, zipper-based version of the pseudocode function "RB-DELETE-FIXUP" [CLR90, p. 274]; the comments show how the various case arms correspond to sections of pseudocode. We therefore focus on the type annotation.

```
(*[ val bbZip :
     -all h, hz : nat-
        zipper(h+1, hz) * rbt(h) → ((bool(true)*rbt(hz - 1)) \/ (bool(false)*rbt(hz)))
    &   BRzipper(h+1, hz) * rbt(h) → ((bool(true)*black(hz - 1)) \/ (bool(false)*black(hz)))
    &   topOrBR(h+1, hz) * black(h) → ((bool(true)*black(hz - 1)) \/ (bool(false)*black(hz)))
```

```
]*)
```

Each part of the intersection shares index refinements; we will look at the first, which has the simplest datasorts:

```
zipper(h+1, hz) * rbt(h) → ((bool(true)*rbt(hz - 1)) \/ (bool(false)*rbt(hz)))
```

Given a zipper that when zipped with a tree of black height $h + 1$ yields a tree of black height hz, and a tree of black height h (one less than $h + 1$, i.e. , with a "black deficit"), `bbZip` returns either

- (`true`, t) where t : rbt(hz−1) (that is, a *valid* tree—with no internal black height mismatches—but with a black height one less than before), or
- (`false`, t) where t : rbt(hz), a valid tree with the same black height hz as the original tree.

The second part of the intersection says that given a zipper that, when zipped with any tree, yields a tree with a black root, the resulting tree (whether of black height $hz - 1$ or hz) will have a black root. The third part of the intersection says that given a zipper that is either TOP or of the kind featured in the second part of the intersection, and a black-rooted tree, the resulting tree must be black. This information is needed when we typecheck `delMin`.

The `delMin` function

`delMin`(t, z) returns the minimum key (an integer) in t; it also returns t with the minimum removed. It calls `bbZip` to fix internal black height mismatches, but like `bbZip` it may be unable to maintain the black height of the entire tree, so like `bbZip` it returns a Boolean indicating whether there is still a "black deficit". The datasorts are needed by `joinRed`, to make sure that the new children of the red node are black.

The functions `joinRed` and `joinBlack`

If one subtree (a or b) is empty, we simply zip up the tree (`bbZip`) with the other subtree; we can drop the first part of `bbZip`'s result—the flag indicating whether the black height has changed—because the zipper z goes all the way up to the original root passed to `delete`; the root of the entire tree has no siblings, so there is no other black height that needs to match.

Otherwise, we call `delMin`, which returns a tree that may or may not have a deficit. If the returned flag is false, there is no deficit, and we can simply zip up to the root. If the flag is true, there is a deficit, and we call `bbZip`—again, throwing away the flag, justified as before.

One quirk is that we have hand-inlined `joinRed`'s call to `delMin`; we could not figure out how to check the color invariant otherwise (there may be some further refinement of `delMin` that does the job, but if there is, it is not obvious). Several of the "inlined" case arms in `delMin` are impossible and disappear, so this only slightly lengthens `joinRed`.

The `delete` function

`delete` and its local function `del` are simple: they search for the key to delete, building a zipper representing the path back up to the root, and then call `joinRed` or `joinBlack`.

Listing 7.5: rbdelete.rml

```
(* rbdelete.rml

      Based on redblack−full.rml and the SML/NJ library file redblack−map−fn.sml
*)
(*[
  datatype  dict with  nat
  datasort  dict :
        badLeft < dict; badRoot < dict; badRight < dict;
   rbt < badLeft;           rbt < badRoot;          rbt < badRight;
                                  nonempty < rbt;   black < rbt;
            red < nonempty;  nonemptyBlack < nonempty; nonemptyBlack < black

    datacon  Empty : black(0)

    datacon  Black :
            -all  h : nat- int * dict(h) * dict(h) → dict(h+1)
                       &  int * rbt(h) * rbt(h) → nonemptyBlack(h+1)
                       &  int * badRoot(h) * rbt(h) → badLeft(h+1)
                       &  int * rbt(h) * badRoot(h) → badRight(h+1)

    datacon  Red :
            -all  h : nat- int * dict(h) * dict(h) → dict(h)
                       &  int * black(h) * black(h) → red(h)
                       &  int * rbt(h) * black(h) → badRoot(h)
                       &  int * black(h) * rbt(h) → badRoot(h)
]*)
datatype  dict =
    Empty
  | Black of  int * dict * dict
  | Red of  int * dict * dict

(*[
  (* z : zipper(h, hz) if zip(z, t) : rbt(hz), where t : rbt(h). *)
  datatype  zipper with  nat * nat

  (* Our datasort refinement captures two distinct properties:

      − the color of the hole's parent:
            If blackZipper, the root (of the zipper, that is, the parent of the hole)
              is black and can therefore tolerate black trees as well as red;

      − the color of the root *of the corresponding zipped tree*:
          When a zipper consisting of various LEFTx and RIGHTx constructors
          applied to a final LEFTB(_,_,TOP) or RIGHTB(_,_,TOP) is actually zipped,
          the resulting tree will be Black(...), and this is represented by the datasort
          BRzipper (Black Root zipper).

          Zipping TOP with a tree 't' results in just the tree 't', so we need to
          distinguish TOP, via datasort topZipper.

          Zipping a black tree and a topZipper yields a black tree;
          zipping a black tree (or a red tree) and a BRzipper yields a black tree;
          Therefore, zipping a black tree and either a topZipper or a BRzipper
          will yield a black tree, so we have a datasort topOrBR that is above both
```

```
            topZipper and BRzipper.

           The datasort blackBRzipper represents zippers that have both a
           black root *of the zipper* and that will have a black root *when zipped*. *)

 datasort  zipper :
                 topOrBR < zipper;                    blackZipper < zipper;
          BRzipper < topOrBR; topZipper < topOrBR;   topZipper < blackZipper;
                 blackBRzipper < topZipper; blackBRzipper < BRzipper

 datacon  TOP : -all  h : nat- topZipper(h, h)

 datacon  LEFTB : -all  h, hz : nat- int * rbt(h) * zipper(h+1, hz) -> blackZipper(h, hz)
                               &   int * rbt(h) * topOrBR(h+1, hz) -> blackBRzipper(h, hz)
 datacon  RIGHTB : -all  h, hz : nat- rbt(h) * int * zipper(h+1, hz) -> blackZipper(h, hz)
                                &   rbt(h) * int * topOrBR(h+1, hz) -> blackBRzipper(h, hz)

 datacon  LEFTR : -all  h, hz : nat- int * black(h) * blackZipper(h, hz) -> zipper(h, hz)
                                 & int * black(h) * blackBRzipper(h, hz) -> BRzipper(h, hz)
 datacon  RIGHTR : -all  h, hz : nat- black(h) * int * blackZipper(h, hz) -> zipper(h, hz)
                                  & black(h) * int * blackBRzipper(h, hz) -> BRzipper(h, hz)
]*)
  datatype  zipper
    = TOP
    | LEFTB of int * dict * zipper
    | LEFTR of int * dict * zipper
    | RIGHTB of dict * int * zipper
    | RIGHTR of dict * int * zipper
;

(*[ val zip :
        -all  h, hz : nat-
              blackZipper(h, hz) * rbt(h) -> rbt(hz)
           & zipper(h, hz) * black(h) -> rbt(hz)
           & blackBRzipper(h, hz) * rbt(h) -> black(hz)
           & topOrBR(h, hz) * black(h) -> black(hz)
]*)
fun zip arg = case arg of
    (TOP, t) => t
  | (LEFTB (x, b, z as _), a)   => zip(z, Black(x, a, b))
  | (RIGHTB(a, x, z as _), b)   => zip(z, Black(x, a, b))
  | (LEFTR (x, b, z), a)        => zip(z, Red(x, a, b))
  | (RIGHTR(a, x, z), b)        => zip(z, Red(x, a, b))

(* bbZip propagates a black deficit up the tree until either the top
 * is reached, or the deficit can be covered. It returns a boolean
 * that is true if there is still a deficit and the zipped tree.
*)
  (*[ val bbZip :
       -all  h, hz : nat-
        zipper(h+1, hz) * rbt(h) -> ((bool(true)*rbt(hz - 1)) \/ (bool(false)*rbt(hz)))
    &   BRzipper(h+1, hz) * rbt(h) -> ((bool(true)*black(hz - 1)) \/ (bool(false)*black(hz)))
    &   topOrBR(h+1, hz) * black(h) -> ((bool(true)*black(hz - 1)) \/ (bool(false)*black(hz)))
  ]*)
```

```
  fun bbZip arg =
      case arg of
         (TOP, t) => (true, t)

    | (LEFTB(x, Red(y, c, d), z), a) =>  (* case 1L—Black *)
       bbZip (LEFTR(x, c, LEFTB(y, d, z)), a)

    | (LEFTB(x, Black(w, Red(y, c, d), e), z), a) =>  (* case 3L—Black *)
       (false, zip (z, Black(y, Black(x, a, c), Black(w, d, e))))
    | (LEFTR(x, Black(w, Red(y, c, d), e), z), a) =>  (* case 3L—Red *)
       (false, zip (z, Red(y, Black(x, a, c), Black(w, d, e))))

    | (LEFTB(x, Black(y, c, Red(w, d, e)), z), a) =>  (* case 4L—Black *)
       (false, zip (z, Black(y, Black(x, a, c), Black(w, d, e))))
    | (LEFTR(x, Black(y, c, Red(w, d, e)), z), a) =>  (* case 4L—Red *)
       (false, zip (z, Red(y, Black(x, a, c), Black(w, d, e))))

    | (LEFTR(x, Black(y, c, d), z), a) =>  (* case 2L—Red *)
       (false, zip (z, Black(x, a, Red(y, c, d))))
    | (LEFTB(x, Black(y, c, d), z), a) =>  (* case 2L—Black *)
       bbZip (z, Black(x, a, Red(y, c, d)))

    | (RIGHTB(Red(y, c, d), x, z), b) =>  (* case 1R—Black *)
       bbZip (RIGHTR(d, x, RIGHTB(c, y, z)), b)

(*(NJ library:) | (RIGHTR(Red(y, c, d), x, z), b) =>  (* case 1R—Red *)
     (* ?!? This is a color violation: RIGHTRed — Red! *)
       bbZip (RIGHTR(d, x, RIGHTB(c, y, z)), b)  (* ...and there is no corresponding arm for 1L *)
*)

    | (RIGHTB(Black(y, Red(w, c, d), e), x, z), b) =>  (* case 3R—Black *)
       (false, zip (z, Black(y, Black(w, c, d), Black(x, e, b))))
    | (RIGHTR(Black(y, Red(w, c, d), e), x, z), b) =>  (* case 3R—Red *)
       (false, zip (z, Red(y, Black(w, c, d), Black(x, e, b))))

(* This 4R is correct —— unlike the buggy NJ library *)
    | (RIGHTB(Black(y, c, Red(w, d, e)), x, z), b) =>  (* case 4R—Black *)
       (false, zip (z, Black(w, Black(y, c, d), Black(x, e, b))))
    | (RIGHTR(Black(y, c, Red(w, d, e)), x, z), b) =>  (* case 4R—Red *)
       (false, zip (z, Red  (w, Black(y, c, d), Black(x, e, b))))

    | (RIGHTR(Black(y, c, d), x, z), b) =>  (* case 2R—Red *)
       (false, zip (z, Black(x, Red(y, c, d), b)))
    | (RIGHTB(Black(y, c, d), x, z), b) =>  (* case 2R—Black *)
       bbZip (z, Black(x, Red(y, c, d), b))
(*
    | (z, t) => (false, zip(z, t))  (* Impossible, bogus fallthru case in NJ library *)
*)


(*[ val delMin :
     -all h, hz : nat-
          nonempty(h) * blackZipper(h, hz)
       -> int * ((bool(false)*rbt(hz)) \/ (bool(true)*rbt(hz - 1)))
   &      nonemptyBlack(h) * zipper(h, hz)
       -> int * ((bool(false)*rbt(hz)) \/ (bool(true)*rbt(hz - 1)))
```

```
   &       nonempty(h) * blackBRzipper(h, hz)
        → int * ((bool(false)*black(hz)) \/ (bool(true)*black(hz - 1)))
   &       nonemptyBlack(h) * BRzipper(h, hz)
        → int * ((bool(false)*black(hz)) \/ (bool(true)*black(hz - 1)))
]*)
fun delMin arg = case arg of
    (Red(y, Empty, b), z) ⇒ (y, ( false(* i.e., no deficit, black height unchanged *), zip(z, b)))

  | (Black(y, Empty, b), z) ⇒
        (* This is the minimum, and it's black, so deleting it yields a black deficit. *)
        (y, bbZip(z, b))
            (* Call bbZip; the flag is important, since it tells the caller whether the resulting
             tree has an internal black deficit (if true) or not (if false).
             (For example, if this is the only non−Empty node in the tree originally passed to delMin,
             there is no way to fix the deficit (nowhere to put the "extra blackness") in this function,
             and the flag will be 'true'.) *)

  | (Black(y, a, b), z) ⇒ delMin(a, LEFTB(y, b, z))
  | (Red(y, a, b), z) ⇒ delMin(a, LEFTR(y, b, z))

(*[ val joinRed : -all h, hz : nat- black(h) * black(h) * blackZipper(h, hz) → rbt ]*)
fun joinRed arg = case arg of
    (Empty, Empty, z) ⇒ zip(z, Empty)
  | (a, Empty, z) ⇒ #2(bbZip(z, a))
  | (Empty, b, z) ⇒ #2(bbZip(z, b))

  | (a, Black(x, Empty, bb), z) ⇒ #2(bbZip(RIGHTR(a, x, z), bb))

  | (a, Black(y, aa, bb), z) ⇒
     let in case delMin(aa, LEFTB(y, bb, TOP)) of
         (x, (needB as  false(* no deficit *), b')) ⇒ zip(z, Red(x, a, b'))
       | (x, (needB as  true(* deficit *), b')) ⇒
           #2(bbZip(RIGHTR(a, x, z), b'))
           (* #2(bbZip(z, Red(x, a, b'))) buggy NJ library: b' could be red (and would not be fixed) *)
      end

(*[ val joinBlack : -all h, hz : nat- rbt(h) * rbt(h) * zipper(h+1, hz) → rbt ]*)
fun joinBlack arg = case arg of
    (a, Empty, z) ⇒ #2(bbZip(z, a))
  | (Empty, b, z) ⇒ #2(bbZip(z, b))
  | (a, b, z) ⇒
      let in case delMin(b, TOP) of
         (x, (needB as false, b')) ⇒ zip(z, Black(x, a, b'))
       | (x, (needB as true, b')) ⇒
             #2(bbZip(RIGHTB(a, x, z), b'))
             (* #2(bbZip(z, Black(x, a, b')))
                (* NJ library version: The black heights of a and b' are different;
                      the Black _ node thus constructed is bogus. *) *)
      end

(*[ val delete : -all h : nat- rbt(h) → int → rbt ]*)
fun delete t key =
    let
      (*[ val del : -all h, hz : nat- rbt(h) * blackZipper(h, hz) → rbt
```

```
                                  &  black(h) * zipper(h, hz) → rbt ]*)
    fun del arg = case arg of
        (Empty, z) ⇒ raise NotFound
      | (Black(entry1 as (key1), a, b), z) ⇒
          if key = key1 then
            joinBlack (a, b, z)
          else if key < key1 then del (a, LEFTB(entry1, b, z))
          else del (b, RIGHTB(a, entry1, z))
      | (Red(entry1 as (key1), a, b), z) ⇒
          if key = key1 then
            joinRed (a, b, z)
          else if key < key1 then del (a, LEFTR(entry1, b, z))
          else del (b, RIGHTR(a, entry1, z))
  in
    del(t, TOP)
  end
```

Listing 7.5: rbdelete.rml

Library bugs

We found two clear bugs in the deletion code in the SML/NJ library; triggering either one results in a tree with a red child of a red parent: that is, the color invariant is broken, making the trees slightly unbalanced. These broken trees are otherwise fine; in particular, they are ordered, so searches succeed or fail as usual, and the failure of the color invariant does not (as far as we could determine) cause subsequent operations to produce disordered trees. Hence, the only calamity caused is that operations will take longer than they should. Since *RedBlackMapFn* makes the exported tree type opaque, client code cannot possibly detect the broken invariant. In that sense, these bugs are pernicious: no one will ever know the module is wrong, unless insertion and deletion are time-critical operations, and they are so stubborn as to actually investigate whether operations in *RedBlackMapFn* are logarithmic. Moreover, runtime testing is not particularly helpful: traversing a tree to verify the invariant is linear time, so adding the tests to every operation makes those operations linear instead of logarithmic, defeating the purpose of using a balanced-tree data structure. (Of course we might dream up more clever tests that would add only constant overhead, but then we have to verify our cleverness.)

The first bug is in the "4R" case of `bbZip`; upon inspection, the case is obviously wrong because it is not symmetric to the "4L" case. We found this bug some time before we converged on the present refinement of `zipper`: we had only a datasort refinement on `zipper`, but even that, combined with actually reading each case closely, sufficed to lead us to this bug.

The second bug is in the `joinRed` function[5]; if `delMin` returns with its first argument true, meaning that the result has a black deficit, the original code calls `bbZip` to fix the deficit; however, the tree passed to `bbZip` includes a red node with `b'` as a child, but `b'` may be red, leading to a color violation (which is *not* somehow fixed inside `bbZip`). We found this second bug much later

[5] The original implementation has a single `join` function rather than separate `joinRed` and `joinBlack` functions; the bug appears to occur in the original code only when `join` is applied with its "color" set to R (meaning Red).

than the first: we had settled on the index refinement of zipper and a nearly-final version of the datasort refinement. Once we became suspicious that `b'` might not always be black, we looked for an input to `delete` that would trigger the bug; we found one, confirming that there was a bug and not simply a case of our invariants, expressed through refinements, being too weak.

We found a few other problems, some of which may also be bugs. One that is not a bug, but still very unfortunate, is the last case arm of bbZip in the original code, which is a "catch-all"

```
| (z, t) ⇒ (false, zip(z, t))
```

This is clearly wrong: `false` is supposed to mean the black deficit was fixed, but the second part of the result is just the argument (as a tree, rather than a zipper with a hole and the tree intended to fill it). For the case to make sense at all, it should return `true`. This is not actually a bug—in our implementation, we leave out the case entirely, and the code still typechecks; thus the case is impossible. Including impossible cases in SML code is an unfortunate but understandable practice, since without type refinements, one would otherwise get a "nonexhaustive match" warning; however, to avoid confusion, the impossible case should be clearly marked, e.g.

```
| (z, t) ⇒ raise Match   (* Impossible case *)
```

7.3 Booleans

If we consider index predicates such as $\geq$ to be index functions, then a Boolean sort manifests itself immediately, as the range of such functions. The Boolean sort can also index the bool datatype. Such an indexing scheme is handy for specifying the result of certain functions. For example, lib_basis.rml defines the type of the ML function < to be

```
primitive  fun < : -all a, b : int- int(a) * int(b) → bool(a < b)
```

If we instead refined bool with a datasort refinement, with sorts false, true $\preceq$ bool, we would have to write

```
primitive  fun < : -all a, b : int-
                    (int(a) * int(b) → bool)
                  & ({a < b} int(a) * int(b) → true)
                  & ({a >= b} int(a) * int(b) → false)
```

If having to write this longer type were the only problem we might not object. However, the datasort refinement formulation also limits the information available when typechecking case arms. For example, it seems we should be able to check that the absolute value function returns a nonnegative integer:

$$(*[\ \ \textbf{val}\ abs : \Pi a{:}\mathcal{Z}.\ \mathsf{int}(a) \rightarrow \Sigma b{:}\mathcal{Z}.\ (b \geq 0 \otimes \mathsf{int}(b))\ \ ]*)$$
$$\textbf{fun}\ abs\ x =$$
$$\quad \textbf{if}\ x < 0\ \textbf{then}\ \tilde{}x\ \textbf{else}\ x$$

The expression **if-then-else** is syntactic sugar for **case** $x < 0$ **of** True $\Rightarrow$ ~x | False $\Rightarrow$ x. With bool refined only by a datasort, typechecking goes as follows. The function application $x < 0$ synthesizes bool (it cannot synthesize true or false since we do not know whether $a < b$ holds or $a \geq b$ holds).

Then in the $\mathsf{True} \Rightarrow \tilde{}x$ arm, we must check that $\tilde{}x$ is nonnegative, i.e. that x is strictly negative. We know that x is indexed by some integer a, but we know nothing about a. Contrast this with the situation when we refine the type bool by the sort bool: now $x < 0$ synthesizes $\mathsf{bool}(a < 0)$, and when we are in the $\mathsf{True} \Rightarrow \tilde{}x$ arm, we will have the assumption $\mathsf{true} \doteq a < 0$.

Finally, the additional intersections in the datasort refinement version can greatly slow type-checking of programs that call <, even if typechecking really does not depend on connecting the arguments of < and its result; the typechecker is not smart enough to know when that connection is irrelevant. (We initially used a datasort refinement of bool and specified the types of <, <>, and so forth with intersections, until we discovered the resulting limitation in power. Faster typechecking was a bonus.)

The structure of the Boolean sort, as implemented, is very simple:

	mathematical notation	Stardust notation
Index sort	bool	`bool`
Constants	true, false	`true`, `false`
Functions	none	
Predicates	none	

We could enrich this with some of the usual Boolean operators (which could be considered functions *and* predicates)—at least conjunction, since that is already present in the constraint language, but we have not encountered any situation calling for this.

7.4 Dimensions: an invaluable refinement

Dimensions are ubiquitous in physics and related disciplines. For example, the plausibility of engineering calculations can be checked by seeing whether the dimension of the result is the expected one. If one concludes that the work done by a physical process is $x \cdot (a_1 + a_2)$ where x is a distance and a_1, a_2 are masses, something is wrong. If, on the other hand, the conclusion has the form $x \cdot (n_1 + n_2)$ where n_1 and n_2 are forces, it is at least possible that the calculation is correct, work being a product of distance and force. The basic operations such as addition are subject to sanity checking through dimensional analysis: one cannot add a distance to a force, and so forth. (*Dimension* refers to a quantity such as distance, mass or time; systems of *units* define base quantities for dimensions. For example, in civilized countries, the base unit of distance is the meter.) Moreover, dimensions aid in the discovery of physical laws: *dimensional analysis* constrains the form of the law by permitting only equations that are dimensionally consistent.

The idea of trying to catch dimension errors in programs is old. Kennedy's dissertation [Ken96] cites sources as early as 1978; Allen et al. [ACL+04], as early as 1975. Many dimension checking schemes were hamstrung by their lack of polymorphism: they could not universally quantify over dimension variables. For example, they could not express a suitably generic type for the square function $\lambda x.\, x * x$. House's extension of Pascal [Hou83] and Wand et al.'s extension of ML [WO91] could express such types, but lacked user-definable dimensions. Kennedy's system, extending Standard ML, is an elegant formulation providing dimension polymorphism and user-definable dimensions. However, Kennedy's formulation is a substantial extension of the underlying type system, and

is complicated by doing full inference rather than bidirectional checking. In our system, dimensions are, formally speaking, just another index domain; practically speaking, the implementation work involved was modest (less than one person-week).

The basic idea is to refine the primitive type real of floating point numbers[6] with a dimension. Certain quantities, including nonzero floating-point literals, are dimensionless and indexed by *1* (though the zero literal 0.0 has type $\Pi a{:}\mathsf{dim}.\,\mathsf{real}(a)$). Constants M, S, and so forth have type $\mathsf{real}(\mathbf{m})$, $\mathsf{real}(\mathbf{s})$, etc.[7] All these constants have the value 1.0, so $3.0 * M$ has value 3.0.

In fact, the value produced by $3.0 * M$ is equal to the values produced by 3.0, and to that produced by $3.0 * S$, by $3.0 * M * M$, and so on. Unlike the data structure refinements used in our examples thus far, dimension refinements say virtually nothing about values! We therefore call it an *invaluable* refinement ("in-" meaning "not", leading to a pun). Zero is an exception to this: it appears that if $\cdot \vdash v : \Pi a{:}\mathsf{dim}.\,\mathsf{real}(a)$ then $v = 0.0$. However, for any $\cdot \vdash v : \mathsf{real}(d)$ (without a Π) the set of possible values is exactly the same for every d, as well as being the same set as the simple type real. This is appropriate—there should be no explicit constructor or tag at runtime.

But what, then, do we actually learn when a program with dimension refinements passes the typechecker? With refinements of lists one could prove properties such that if a closed value has the emptylist refinement it must be Nil, but with dimensions there are few directly corresponding properties. Instead, being well typed means that subterms of dimension type are used in a consistent way. We will explore this notion of consistency first by example, showing which manipulations of dimension-typed entities are considered consistent, and later more formally (Section 7.4.3).

7.4.1 Consistency and casting

The user must make *some* initial claims about dimensions (otherwise everything will be dimensionless and nothing is gained), and these claims cannot be checked, though we can check the consistency of their consequences. For example, the user must be free to multiply by constants such as M, to assign dimensions to literals and to the results of functions like *Real.fromString*. Given free access to those constants, for any *known constant* dimensions d_1, d_2, it is trivial to write the appropriate 'coercion', such as this one for converting $\mathbf{m}^2$ to $\mathbf{kg}$:

$$(*[\ \ \textbf{val}\ \mathit{m2_to_kg} : \mathsf{real}(\mathbf{m} \mathbin{\hat{}} 2) \to \mathsf{real}(\mathbf{kg})\ \ ]*)$$
$$\textbf{fun}\ \mathit{m2_to_kg}\ x = (x/(M * M)) * KG$$

However, we cannot write a 'universal cast' between arbitrary dimensions, unless the dimension constants are also passed to the cast as arguments:

$$(*[\ \ \textbf{val}\ \mathit{universal_cast} : \Pi d_1, d_2{:}\mathsf{dim}.\,\mathsf{real}(d_1) \to \mathsf{real}(d_2)\ \ ]*)$$
$$\textbf{fun}\ \mathit{universal_cast}\ x = \mathit{???}$$

$$(*[\ \ \mathit{cast_with_args} : \Pi d_1, d_2{:}\mathsf{dim}.\,\mathsf{real}(d_1) \to \mathsf{real}(d_1) \to \mathsf{real}(d_2) \to \mathsf{real}(d_2)\ \ ]*)$$
$$\textbf{fun}\ \mathit{cast_with_args}\ x\ u1\ u2 = (x/u1) * u2$$

[6] Like Standard ML and many languages before it, we give the name real to values that are anything but.

[7] Writing M for the meters constant clashes with the established usage of $M/L/T$, etc. for generic dimensions of mass/length/time; see, for example, Logan [Log87]. We do not use generic dimensions (instead we use units in a particular system, namely SI, the "metric system") in this work, so there is no internal inconsistency; anyway, an identifier appearing in a term is unlikely to denote a generic dimension.

As a consequence, there is no useful *Real.toString* of type $\Pi d_1{:}\mathsf{dim}.\,\mathsf{real}(d_1) \to \mathsf{string}$. But we can readily write one *toString* function for each dimension—appending the appropriate string representation of the dimension to the result of *Real.toString* : $\mathsf{real}(1) \to \mathsf{string}$, as in the following.

```
(*[ val meters_toString : real(m) → string ]*)
fun meters_toString x = (Real.toString (x/M)) ^ " m"
```

We could provide a *universal_cast* function as a primitive. If such a function seems horrific, consider the fact that ML implementations typically have library modules that break type safety, such as SML's *Unsafe* and OCaml's *Obj*. However, it is rarely helpful and even more rarely required to use these modules; perhaps most importantly, it is considered unseemly. As long as similar cultural practice prevailed with a *universal_cast* for dimensions, the consequences of including it would be limited. It would be trivial to declare such a function in lib_basis.rml, and the SML implementation would just be the identity, **fun** *universal_cast* $n = n$. Nonetheless, we see no compelling reason to bother.

7.4.2 Definition of the index domain

The structure of the index domain is defined by its index sorts, expressions, and predicates; however, dimensions have no predicates besides index equality. Again, *1* stands for the multiplicative identity that indexes dimensionless quantities.

	mathematical notation	Stardust notation
Index sort	dim	`dim`
Constants	*1*, **m**, **s**, **kg**	`NODIM`, `M`, `S`, `KG`
Functions	$*$, ^	`*`, `^`
Predicates	$\doteq$	`=`

'$*$' is a function of two dimensions (e.g. $\mathbf{m} * \mathbf{s}$), while '^' is a function of a dimension and an integer (e.g. $\mathbf{s} \,\hat{}\, (-1)$).

Choice of exponents

Kennedy [Ken96, p. 7] argues that only integer exponents should be permitted, because fractional exponents have no physical basis. If a fractional exponent appears, say $\mathbf{h}^{1/2}$, this means that in fact the system of units should be changed: **h** should be replaced with its square root, and uses of $\mathbf{h}^n$ should be replaced with $\mathbf{h}^{2*n}$; the old $\mathbf{h}^{1/2}$ then becomes **h**. For example, if we had area as a base dimension **A**, the proper response to the appearance of $\mathbf{A}^{1/2}$ should be to introduce a distance dimension and define area as its square. The counterargument is that one may wish to temporarily create a value whose dimension is physically nonsensical (fractional), on the way to producing a sensible result. We follow Kennedy and permit only integers. However, our constraint solvers support rational arithmetic, so allowing rational exponents should not be difficult.

7.4.3 Soundness

Our type safety theorem does not immediately apply. The first thing is to extend the typing rules and operational semantics to handle real. Let $N^{d}_{\mathbf{R}}$ stand for a real (floating point) number N of dimension d. These *dimension superscripts* have no runtime representation in actual code; we put them in the operational semantics so we can state a soundness conjecture. Gloss literal reals as $N^{1}_{\mathbf{R}}$; for example, 1.5 means $1.5^{1}_{\mathbf{R}}$. Moreover, gloss unit constants such as *M* as $1.0^{\mathbf{m}}_{\mathbf{R}}$, etc.[8]

Add the following rules for typing and transitions:

$$\frac{}{\Gamma \vdash N_{\mathbf{R}} : \mathsf{real}(1)} \qquad \frac{}{\Gamma \vdash 0.0_{\mathbf{R}} : \Pi a{:}\mathsf{dim}.\,\mathsf{real}(a)} \qquad \frac{}{\Gamma \vdash N^{d}_{\mathbf{R}} : \mathsf{real}(d)}$$

$$\begin{array}{rcll} (N_1)^{d_1}_{\mathbf{R}} * (N_2)^{d_2}_{\mathbf{R}} & \mapsto & (N_1 \cdot N_2)^{d_1 \cdot d_2}_{\mathbf{R}} & *_{\mathbf{R}} \\ (N_1)^{d}_{\mathbf{R}} + (N_2)^{d}_{\mathbf{R}} & \mapsto & (N_1 + N_2)^{d}_{\mathbf{R}} & +_{\mathbf{R}} \\ & \vdots & \text{(rules for } -, /, \text{ etc.)} & \end{array}$$

Note that this is no mere decoration, but a restriction of the unrefined semantics: in a dimension-free semantics, the + rule would have the form

$$(N_1)_{\mathbf{R}} + (N_2)_{\mathbf{R}} \mapsto (N_1 + N_2)_{\mathbf{R}}$$

In contrast, rule $+_{\mathbf{R}}$ can be applied only if N_1 and N_2 have the same dimension d. This raises the possibility that progress (not only preservation of dimension refinements) could somehow fail. However, we conjecture that type safety continues to hold for the resulting semantics.

Conjecture 7.1. *If* $\Gamma \vdash e : C$ *(allowing the typing rules for real values above) and* σ *is a substitution over program variables such that* $\vdash \sigma : \Gamma$ *and* $\overline{\Gamma} = \cdot$*, then either*

(1) e value *and* $\vdash [\sigma]\, e : C$*, or*

(2) there exists e' *such that* $[\sigma]\, e \mapsto e'$ *(allowing the transition rules above) and* $\vdash e' : C$.

7.4.4 Implementation

> In most cases this should require the addition or modification of well under a thousand lines of (high-level) source code. (This is not meant to imply that the task is trivial: the lines of code have to be the right ones, and they have to be in the right places.)
>
> —G. Baldwin

In this section, we describe how we implement the dimensions index domain. Neither ICS nor CVC Lite directly support dimensions, so Stardust reduces constraints on dimensions to constraints on integers, as described below.

[8]Equivalently, we could provide user-level syntax for $N^{d}_{\mathbf{R}}$, eliminating unit constants entirely. However, that would depart significantly from SML syntax.

Base dimensions and unit constants

Both the base dimensions **m**, **s**, etc. and the associated unit constants such as *M* are rendered in uppercase and declared in lib_basis.rml as follows:[9]

```
indexconstant  NODIM : dim   (* The only place NODIM should need to be written
                                is in this file, as the default index for type
                                'real', so its ugliness is not a problem. *)
indexconstant  M : dim
indexconstant  S : dim
indexconstant  KG : dim
  ...
indexfun  * : int * int → int, dim * dim → dim
indexfun  / : int * int → int, dim * dim → dim
indexfun  ^ : dim * int → dim
  ...
primitive   type real with dim = NODIM
  ...
primitive   val M : real(M)
primitive   val S : real(S)
primitive   val KG : real(KG)
  ...
```

The dimension *1* is rendered as NODIM; this is rather ugly, but since the type real with no index given is translated to real(NODIM) by the Inject phase (Section 6.3.2), there is no need for the user to write it.

Solving constraints

Stardust reduces constraints on dimensions to a conjunction of constraints on the exponents: $\mathbf{m}^a \doteq (\mathbf{m} * \mathbf{s})^b$, which is equivalent to $(\mathbf{m}^a) * (\mathbf{s}^0) \doteq (\mathbf{m}^b) * (\mathbf{s}^b)$, reduces to $(a \doteq b) \wedge (0 \doteq b)$. Without existentials, that would be the end of the story, since every index expression of dimension sort can be reduced to a normal form in which each base dimension or (*universally* quantified) dimension variable appears once and in some particular order [Ken96, pp. 16–17]. Then equality is just the conjunction of equalities of exponents. Existentials require that we actually solve for dimension variables, but this is not at all difficult.

Given a normal form equation $i_1 \doteq i_2$ containing a factor $\widehat{a}$ (with nonzero exponent), we first rearrange the equation into the form $1 \doteq (i_1^{-1}) * i_2$ and distribute the -1 over the factors in i_1, yielding an equation $1 \doteq \widehat{a}^k * j_1^{k_1} * \cdots * j_n^{k_n}$, where $k \neq 0$. Multiplying both sides by $\widehat{a}^{-k}$ yields $\widehat{a}^{-k} = j_1^{k_1} * \cdots * j_n^{k_n}$; raising both sides to the power $1/(-k)$ gives the solved form $\widehat{a} = \ldots$.

7.4.5 Related work on dimension types in ML

We point out certain differences between Kennedy's work on dimension types in ML and ours. Kennedy [Ken96, p. 66] notes that the function *power* : int → real → real, such that *power* n x

[9] The complete lib_basis.rml appears in Listing 6.1.

yields x^n, cannot be typed in his system; his system lacks any sort of dependent type on integers. With our integer index refinement, this is no problem:

$$power : \Pi d{:}\mathsf{dim}.\, \Pi n{:}\mathcal{Z}.\, \mathsf{int}(n) \to \mathsf{real}(d) \to \mathsf{real}(d \,\hat{}\, n)$$

This function is in kennedy66.rml.

```
(*[ val power : -all d : dim- -all n : int- int(n) -> real(d) -> real(d^n) ]*)
fun power n x =
  if  n = 0 then
    1.0
  else  if  n < 0 then
    1.0 / power (~n) x
  else
    x * power (n-1) x
```

Similarly, in Kennedy's system, universal quantifiers over dimension variables must be prenex (on the outside), just like universal quantifiers over SML type variables. Kennedy [Ken96, pp. 66–67] gives the example of a higher-order function *polyadd* that applies prod to arguments of different dimensions (first to *1* and **kg**, then to **kg** and *1*); Kennedy's system cannot infer the type *polyadd* : $(\Pi d_1{:}\mathsf{dim}.\, \Pi d_2{:}\mathsf{dim}.\, \mathsf{real}(d_1) \to \mathsf{real}(d_2) \to \mathsf{real}(d_1 * d_2)) \to \mathsf{real}(\mathbf{kg})$ because the Πs are inside the arrow.

This function is in kennedy67.rml.

$$\textbf{fun}\ \textit{polyadd}\ \texttt{prod} = \texttt{prod}\ 2.0\ \textit{KG} + \texttt{prod}\ \textit{KG}\ 3.0$$

Since we do not require universal index quantifiers to be prenex, we have no difficulty typechecking *polyadd*.

We close with two additional examples from Kennedy [Ken96, p. 11]. The first implements the Newton-Raphson method.

This function is in kennedy11.rml.

```
(*[ val newton : -all d1,d2:dim-
  (* f, a function *)             (real(d1) -> real(d2))
  (* f', its derivative *)       * (real(d1) -> real((d1 ^~1) * d2))
  (* x, the initial guess *)     * real(d1)
  (* xacc, relative accuracy *)  * real
                                -> real(d1)
]*)
fun newton (f, f', x, xacc) =
  let  val dx = f x / f' x
       val x' = x - dx
  in
       if  abs dx / x' < xacc
       then x'
       else  newton (f, f', x', xacc)
  end
```

The second illustrates how, while Kennedy's inference algorithm must actually determine the least common multiple of 2, 5, and 6 (the number of factors in each term in the function), bidirectional checking means that we simply do trivial arithmetic on small integers ($15 + 15 = 30$, etc.).

This function is in kennedy11b.rml.

```
(*[ val powers : -all d:dim- real(d^15) * real(d^6) * real(d^5) → real(d^30)
]*)

fun powers (x, y, z) = x*x + y*y*y*y*y + z*z*z*z*z*z
```

Kennedy also presents results about dimension polymorphism that are in the same vein as Wadler's "theorems for free" [Wad89] based on Reynolds' work on parametricity [Rey83], and discusses how Buckingham's Π Theorem [Lan51, Log87]—about the behavior of equations under a change of dimensions—might carry over to dimension types. We do not know if similar results hold for our system.

7.4.6 Units of the same dimension

Some of the most catastrophic dimension bugs are not strictly attributable to confusion of dimensions, but to confusion of *units*. In 1984, the space shuttle Discovery erroneously flew upside down because a system was given input in feet, when it expected input in nautical miles [Ken96, p. 12].[10] And in 1999, NASA's $125 million Mars Climate Orbiter was lost and presumed destroyed after navigational errors. Initial reports put the blame on unit confusion—the result of a calculation based on Imperial units was given to onboard software expecting quantities in SI units. While some have argued that the causes of the loss were multifarious [Obe99], it is clear that unit confusion played a role: an "impulse-bit" in pound-forces was not converted to newtons. The software performing this calculation was a modified version of software for an earlier space probe; even though the conversion *was* done correctly in the original software, the conversion was not obvious in the code, and was lost when the system was revised for the Mars Climate Orbiter [EJC01, p. 7].

Our system does not directly support multiple units of the same dimension. We see two ways to handle programs with mixed units:

1. Recognize units of the same dimension, such as **ft** and **m**, and automatically convert among them. For example, if we have feet **ft** as a unit of distance, applying a function of type $\mathsf{real}(\mathbf{ft}) \rightarrow \mathsf{bool}$ to an argument of type $\mathsf{real}(\mathbf{m})$ would be legal, and the compiler would insert a coercion (multiplying by 0.3048) to convert meters to feet.

2. Consider units of the same logical dimension to be distinct dimensions. Conversion of feet to meters would not be automatic; the user would have to explicitly call a conversion function, as in the example in Figure 7.3.

The first way needs compiler support (and of course a proper theoretical foundation), and is therefore outside the scope of our current implementation approach, but there is a more fundamental objection: it is already difficult enough to reason about floating-point computations without the compiler sneaking in unit conversions. Most of the time, perhaps, the hidden conversions would be harmless, but over- or underflow is conceivable. If we followed the second way, the conversions

[10] Kennedy, who quotes a report of the incident in ACM SIGSOFT Software Engineering Notes vol. 10 no. 3 (July 1985), says this was "really a problem with the human-computer interface". Fair enough, but in implementing the obvious fix—to refuse input of unadorned numbers, requiring instead that the units be appended to the string (as "`10023 ft`")—one would write conversion functions whose interface (and trivial implementation) *could* be usefully checked with dimension/unit types.

```
(*[
  indexconstant  M : dim
  indexconstant  FT : dim
  ...
  primitive  val M : real(M)
  primitive  val FT : real(FT)
]*)

(*[  val metersPerFoot : real(M / FT)   ]*)
val metersPerFoot = (0.3048 * M) / FT

(*[  val fromFeet : real(FT) → real(M)  ]*)
fun fromFeet ft = ft * metersPerFoot

(*[  val toFeet : real(M) → real(FT)   ]*)
fun toFeet m = m / metersPerFoot

(*[  val parse : string → real * string ]*)
...
(*[  val readDistance : real → string → real(M) ]*)
fun readDistance number units =
  case units of
      "m" ⇒ number * M
    | "ft" ⇒ fromFeet (number * FT)
    | _ ⇒ raise BadInput ("unknown␣distance␣unit:␣"^units)
```

Figure 7.3: Units of the same dimension in Stardust

would be obvious. We expect (or hope) that in most systems with mixed units, large components do *not* mix units; for instance, if feet appear in input data but all interesting computation is in meters, the burden of manually adding a few conversions would be small. The second strategy also has the virtue of requiring no changes to the index domain.

A variant of the first approach was followed by Allen et al. [ACL+04] in their work on dimensions and units in an object-oriented language with nominal typing. They significantly extend the Java language with "metaclasses", "instance classes", and "abelian classes", of which dimensions are an instance.

7.4.7 Related work on invaluable refinements

Our term "invaluable refinement" is new, but similar refinements have come up in other contexts. In security, untrusted data is considered *tainted*; for instance, a string literal in a program should be safe to use on a command line passed to the OS function `system`, whereas a string entered into a Web form is considered "tainted" and should not, in most circumstances, be passed to `system`. Similarly to dimensions, the sets of tainted and untainted values are identical. "Casting" between tainted and untainted data is permitted; the idea is to make such casts obvious.[11] "Taint checking" can of course be performed at runtime, but static checking has been investigated by Foster [Fos02], whose *qualified types* encompass a variety of flow-sensitive, invaluable properties[12]: Zero or more qualifiers, under a partial order (reminiscent of datasort refinements), may appear with a type. Foster's qualified type annotations are of two forms: $\texttt{annot}(e, Q)$, which adds the qualifier Q to the type inferred for e (a kind of cast), and $\texttt{check}(e, Q)$, which directs the system to check that Q is among the qualifiers of the type inferred for e. Thus, as with dimensions in our system, qualified types are based on annotations provided by the user and cannot be checked at runtime. In our system, we suspect that either a datasort refinement or an index refinement with a domain of finite sets of constants (the qualifiers) would suffice to model qualified types, with some kind of cast—some well-named identity function—acting as $\texttt{annot}(e, Q)$, and type annotation $(e : A)$ with the appropriate refinement acting as $\texttt{check}(e, Q)$. If an index refinement were chosen, the type A might take the form $\Sigma s{:}\mathsf{quals}.\,(Q \in s) \otimes \tau(s)$. For example, the type of strings qualified by *Tainted* might be $\Sigma s{:}\mathsf{quals}.\,(\mathit{Tainted} \in s) \otimes \mathsf{string}(s)$.

In a curious way, garden-variety Hindley-Milner typing also supports invaluable refinements. A *phantom type* [FLMP99, LM99] is merely an ordinary datatype with a "phantom" polymorphic type parameter that is not tied to the values of that type, at least not in the completely obvious way of the α in α list. For example, we can cook up our own tainted/untainted string datatype in SML by defining dummy opaque types tainted and untainted, used only in the polymorphic type parameter of the datatype α str:

[11] Perl, which has a notion of tainted data (checked at runtime), fails to get even this right: the standard way to "launder" an entire string is not to call some suggestively named identity function but to copy the string to a new variable by matching the tainted string against a regular expression that matches everything; the new string will be untainted.

[12] Foster's work, and ours, is distinct from work focused on statically ensuring that high-security information does not leak to low-security processes (e.g. [Mye99, Zda02, CKP05]), which considers *only* security levels, providing strong guarantees (e.g. noninterference) over a different domain.

```
signature STR = sig                      structure Str :> STR = struct
  type tainted                             type tainted = unit
  type untainted                           type untainted = unit
  type 'a str                              datatype 'a str = Str of string
  ...                                      ...
  val untaint :                            fun untaint (Str s) =
        tainted str → untainted str          Str s
end                                      end
```

A few points should be noted. First, the actual definitions of the dummy types tainted and untainted are irrelevant since no values of these types are actually created. Second, the *untaint* function merely re-creates the underlying value, since a newly constructed Str can have whatever α we like—in this case, untainted.

More complicated uses of phantom types can even mimic integer index refinements by encoding integers through dummy types, a technique Blume uses in his "No Longer Foreign Function Interface" for Standard ML of New Jersey [Blu01]. Blume uses phantom types to model the lengths of C-language arrays, which might seem to be a value-based refinement rather than an invaluable one; however, from the standpoint of the SML type system it is invaluable, since C arrays are not accounted for in SML's semantics. From the user's point of view, integer index refinements seem more natural.

Phantom types can also be used as value refinements, but the typechecker's ability to reason based on inversion is limited. Our intention in the "taint" example above is that α str will be instantiated only by tainted or untainted, not by int or string $\rightarrow$ string, etc. However, even if we are careful to only instantiate α str as we intend (which is not difficult, since str is abstract outside the structure Str) the typechecker does not know this is the case. For this invaluable refinement, there are no inversion principles (not counting "any value of type A str where A is a type must have the form Str s", which we knew anyway!), but when phantom types are used to encode some value-based property, such as the length of an SML list, this is a serious shortcoming, especially since exhaustiveness of pattern matching cannot be shown. Hence, researchers have designed various systems supporting "first-class" phantom types, whether called that [CH03, FP06], *guarded recursive datatypes* [XCC03], *generalized algebraic datatypes* [PVWW06], or *equality-qualified types* [SP04]. This approach eliminates one virtue of phantom types: that one can use any standard compiler for SML, Haskell, etc.[13]

From our (biased) perspective, phantom types (whether first- or second-class) are tantamount to index refinements in which the index objects are types. These systems do not have intersection types and cannot express conjunctions of refinement properties, at least not trivially. Perhaps more fundamentally, when the index objects are types, index equality is type equivalence—which, as equational theories go, is rather impoverished in conventional type systems. It is no coincidence that one of the standard examples of phantom types is an interpreter for a tiny typed language, where (in our terminology) terms in the interpreted language are indexed by types. There, the encoding from the problem domain—types of terms in the interpreted language—is trivial, because arrows, products, etc. are among the types of the source language. This encoding breaks down if

[13] This will likely be less of an issue in the future; in fact, first-class phantom types are now supported in the Glasgow Haskell Compiler [PVWW06].

the types of the interpreted language are in some significant way more expressive than the types of the source language.

To obtain richer index domains (in our terminology) than their current type expressions, some have enriched phantom type systems with elements of traditional dependent typing [She04]. These systems allow the user to write their own proofs of properties in undecidable domains, an ability missing in our system. From a user's perspective, this approach seems more complex than ours. However, given the present work's conspicuous lack of parametric polymorphism, we would be unwise to cast any more aspersions.

Ephemeral refinements [MWH03] may be a form of invaluable refinement as well: the ephemeral refinements are about 'the state of the world', which is not a directly manipulable value in SML and many related type systems. If we consider only ephemeral refinements involving mutable storage, a monadic formulation of ephemeral refinement systems would reify the state into a value and the ephemeral/invaluable refinement of the state into a value refinement. Think of Haskell's state monad with a refinement about the array's contents: the contents of the array become part of the world encapsulated by the monad. However, given an ephemeral refinement that encodes information that cannot be directly inspected, such as (some property of) the string representing the program's output so far, there is nothing to reify; unless the program is modified (not merely annotated) to store the information so it can be inspected, there is no value to refine. Thus, both value and invaluable refinements should be useful in a system with monadic encapsulation of effects.

We end by observing that, as implied in Section 4.9.2, Davies' datasort refinement system does not support invaluable refinements. In his system, the inhabitants of the datasorts are specified through regular tree grammars in which the symbols are the datatype's constructors; the only way to define datasorts that are not perfectly synonymous is to specify that they are inhabited by different sets of values. (Serendipitously, mere laziness kept us from following the same strategy: we simply did not want to bother transforming regular tree grammar-based specifications into signatures $\mathcal{S}(\mathsf{c})$!)

7.5 Conclusion

We have formulated index domains of integers, dimensions, and Booleans, and implemented them in Stardust. Our examples illustrate the power of these domains, as well as certain limitations, such as our difficulty with `add` in the bitstrings example. That difficulty may be surmountable, but users have finite time and effort to put into making refinements work. However, in programming (unlike pure mathematics), the perfect should not be allowed to be the enemy of the good. Completely correct programs can be achieved only at great, and often unwarranted, cost.

Besides dimensions, other invaluable refinements may prove a rich lode. Because they are not bound up in the details of data structures, they seem attractively simple.

As the set of supported domains grows, an already present problem grows with it: the scalability of the refinements themselves. Different invariants will be important in different parts of a program. It is perfectly reasonable to index lists by length; it is also perfectly reasonable to index them by their contents, or by some property of the first or second or ith element. Our current approach requires that one either cram all manner of indices into a tuple, and index by that, or create new

datatypes for each new property, each with its own refinement. The first technique is brazenly anti-modular; the second leads to code duplication and tedium. Thus, designing modular refinements is an important goal for future work. We mention here only a preliminary idea. First, we must permit the *post hoc* addition of datasort and/or index refinements to a previously declared type. By itself, that only allows us to avoid writing out the same datatype more than once; the functions manipulating that datatype will not "know" about the new refinements. So we must also allow the *post hoc* modification of function types. This should be safe (and not break code using the old refinement) if the modified type principally synthesizes the original type; in particular, if we already have $f : A_1 \to A_2$ with A_1 and A_2 in the "old" refinements, we can add an intersection to get $f : (A_1 \to A_2) \wedge (B_1 \to B_2)$, with B_1 and B_2 using the new refinements—and $(A_1 \to A_2) \wedge (B_1 \to B_2) \uparrow (A_1 \to A_2)$, so we can still typecheck code depending on f synthesizing $A_1 \to A_2$. Of course, when we modify the type of f in this way, we must check it against the new conjunct $B_1 \to B_2$.

Chapter 8

Conclusion

> **Conclusion** I did it and it works.
>
> —G. Baldwin

> ... we know from past experience that what has been sufficiently expensive is automatically declared to have been a great success.
>
> —Edsger W. Dijkstra

We have formulated a type system that is rich, yet practical. We have gone beyond work on various refinements in isolation to a combination of atomic refinements. Our work builds on Davies and Pfenning's illuminating, practical approach to intersection types, and goes beyond Xi and Pfenning's work on index refinements by combining them, for the first time, with intersections. We have formulated an elegant theory of marker-free unions and index-level existentials, made practical through our let-normal transformation. Index refinements are parametric in the index domain, but until now, the integer domain has ruled. Our work on dimensions breaks past that, and demonstrates that our approach to refinements can check properties that are not value-based.

There are several key elements underlying our work. One is bidirectional typechecking. We thus rely on information given by the user rather than the result of some bottom-up analysis. This does not only yield an elegant theory, but seems highly effective for programs in which the user understands what is going on. Those are the good programs—there will always be mistakes and errors as the program is developed, but we can catch many of them with the aid of the user's specifications. Unlike many other approaches to catching bugs, we can reasonably hope that when a program passes typechecking it really does have the types claimed. Of course there will be bugs in typecheckers, constraint solvers, and so forth; we are still in the gutter, but we are looking up at the stars.

Another element is respect for the subformula property. Though never precisely formulated or proved for our systems, it influenced important design decisions; it is one reason that we have no distributivity rules in our subtyping system, and that we do not intersect typings in contextual typing annotations. It may also be why our implementation is as successful as it is: the typechecker's speed would be grossly unpredictable if the typechecker fabricated backtracking-inducing property types.

A final element of our type refinements is usability. We believe that types with datasort and index refinements are easy to write and understand, in contrast to approaches closer to true dependent types (the name Dependent ML aside, index refinements are a pale shadow of dependent types, and we like them that way). Such approaches have their place, but we would rather build up from the ground than dive from the sky.

The remainder of this chapter examines a number of avenues for future work.

8.1 Future work

8.1.1 Parametric polymorphism

The lack of parametric polymorphism is a major limitation of the present system. Adding parametric polymorphism to the type assignment system of Chapter 2 appears rather straightforward: add type variables α, β and a universal quantifier over types (not, like Π, over indices) $\forall\alpha.\, A$ to the grammar of type expressions, extend Γ to allow assumptions of the form α type, and add the following subtyping and typing rules[1]:

$$\frac{\Gamma \vdash [A'/\alpha]A \leq B \qquad \overline{\Gamma} \vdash A' \;\mathsf{ok}}{\Gamma \vdash \forall\alpha.\, A \leq B}\,\forall\mathrm{L} \qquad \frac{\Gamma, \beta \vdash A \leq B}{\Gamma \vdash A \leq \forall\beta.\, B}\,\forall\mathrm{R}$$

$$\frac{\Gamma, \alpha\;\mathsf{type} \vdash v : A}{\Gamma \vdash v : \forall\alpha.\, A}\,\forall\mathrm{I} \qquad \frac{\Gamma \vdash e : \forall\alpha.\, A \qquad \Gamma \vdash A' \;\mathsf{ok}}{\Gamma \vdash e : [A'/\alpha]\, A}\,\forall\mathrm{E}$$

For a type assignment system, decidability is of little concern. We are inclined to believe, but have not proved, that the results shown for the system in Chapter 2 hold for the system with these new rules; the key results are substitution, value definiteness, and type safety. We suspect that the interesting problems lie further on, as we move from type assignment to the tridirectional framework. Figuring out the judgment directions for ∀I and ∀E seems easy enough, following the principle that introduction rules check and elimination rules synthesize that we used successfully in Chapter 3:

$$\frac{\Gamma, \alpha\;\mathsf{type} \vdash v \downarrow A}{\Gamma \vdash v \downarrow \forall\alpha.\, A}\,\forall\mathrm{I} \qquad \frac{\Gamma \vdash e \uparrow \forall\alpha.\, A \qquad \Gamma \vdash A' \;\mathsf{ok}}{\Gamma \vdash e \uparrow [A'/\alpha]\, A}\,\forall\mathrm{E}$$

∀I is straightforward enough, but with ∀E (and ∀L) we have to somehow "guess" the type A' that replaces α. With ΠE we had a similar problem of guessing an index; as discussed in Chapter 6 we solved that by introducing an existential index variable and solving for it. Often no solution is known when ΠE is applied, so typechecking simply goes on with the existential index variable "in tow". This seems intuitively valid because indices exist on a level distinct from (and inferior to) types. With ∀E everything is at the type level, and caution is advised: inference is undecidable for many systems with similar features, including System $F_{<:}$ [Pie94], even without distributivity [Chr98].

[1]For simplicity, we do not treat polymorphic datatypes (e.g. list would be covariant in the type argument, so α list $\leq$ β list if $\alpha \leq \beta$), as a realistic proposal would require.

The key problem, then, is figuring out how to instantiate polymorphic $\forall\alpha.\, e$ types. We see several possible approaches.

- *With a refinement restriction, do simple type inference first*. The *refinement restriction* found in past work on (datasort) refinements disallows $A \wedge B$ (and $A \vee B$) if A and B are not refinements of the same simple (unrefined) type. In the bitstrings refinement, $\mathsf{bits} \wedge \mathsf{pos}$ would be acceptable, but $\mathsf{bits} \wedge (\mathsf{pos} \to \mathsf{pos})$ would be invalid.[2] Standard Hindley-Milner inference can find simple polymorphic instances. If the number of refinements of a simple type thus found is finite *and* small, one can enumerate them and try them all; for example, if bits is refined only by a datasort, the typechecker can instantiate it first with bits, and if that fails, with std, and finally with pos. If the number of refinements is large ($\mathsf{bits}*\mathsf{bits} \to \mathsf{bits}*\mathsf{bits}$ has $3^4 = 81$ refinements, for instance), try the simple type alone (e.g. bits); if typechecking fails, the user must annotate.

  This is essentially how Davies' typechecker works; lacking index refinements, he always has a finite number of refinements but must handle the possibility that there are too many [Dav05a, pp. 241–243]. However, in our system, anything indexed by a sort with infinitely many index elements, such as the integer sort $\mathcal{Z}$, has infinitely many refinements.

- *Types with modes*. Davies also discusses allowing the user to specify directional modes of types [Dav05a, pp. 241–243]. The typechecker can be told that the argument to a function $g : \forall\alpha.\, \alpha \to B$ should be inferred, so that when g is applied to e a type is synthesized for e (rather than checked as rule $\to$E would have it), yielding the instantiation for α. One possible syntax would be $^{\uparrow}\!\!\to$ for arrows in which the argument is to be synthesized:

  $$g : \forall\alpha.\, \alpha \;^{\uparrow}\!\!\to B$$

  This does not work well with multiple occurrences of α. Suppose we have $f : \forall\alpha.\, \alpha \to \alpha \to \alpha$, $x : A$, $y : B$, and the application $(f\ x)\ y$. An intersection might allow either the first or second argument to synthesize:

  $$f : \forall\alpha.\, (\alpha \;^{\uparrow}\!\!\to \alpha \to \alpha) \wedge (\alpha \to \alpha \;^{\uparrow}\!\!\to \alpha)$$

  Right away we have a problem, because in the second part $\alpha \to \alpha \;^{\uparrow}\!\!\to \alpha$ we must somehow skip the first argument so we can get to the second. But leaving that aside, we still cannot handle the case where neither $A \leq B$ nor $B \leq A$.

  Another objection to this approach is that one can no longer see immediately which terms are synthesizing and which are checking, since that is now dependent on types rather than mere syntactic forms. That is, one must know the type of g to know whether e in $g\ e$ needs to be a synthesizing form; if e is a checking form, an annotation must be added. While some other bidirectional formulations, such as Xi's [Xi98], are more liberal than ours—Xi allows (x, y) to synthesize since x and y do, for instance—directionality is still a *local* syntactic property, just deeper than in our system, where directionality is based solely on the root syntactic form.

[2] Of course, SML compilers cannot do anything sensible with expressions of the latter type—but such types are perfectly valid in our type system.

That is, in our system, the user can deduce a term's directional character by looking at the root of its abstract syntax tree; in Xi's, by looking at the whole abstract syntax tree of the term; but if directional modes are permitted in declared types, one must look at the entire program.

We could modify the approach by moving the directional marker from the type to the use: instead of $g\ x$, write $g\ \ \$^{\uparrow}\ \ x$. This avoids the last objection, at the cost of invading term syntax with a new form of annotation.

- *Local type inference*. Pierce and Turner [PT98] describe a strategy for type inference with subtyping and (impredicative) polymorphism. Their strategy has two components: bidirectional typing and local instantiation of polymorphic function applications. The first is not radically unlike bidirectionality in our system. The second uses a constraint-generating system. The main idea is that the constraints are manipulated to yield polymorphic instance(s) *locally*, i.e. within that application of a polymorphic function. Their system requires construction of greatest lower and least upper bounds of types, going to some effort to ensure that such bounds actually exist. That problem is trivial in our system, where those bounds are simply intersection and union, respectively—and since users can write intersections and unions, a system modeled on ours should not be expected to make them up on its own.

- *Be greedy*. Suppose we resign ourselves to towing along an existential type variable. A strategy of accumulating constraints (type equations and subtypings) and manipulating them, nonlocally, seems too powerful. It also seems highly nontrivial, given the presence of intersections and unions. But what if we keep the accumulated constraints very simple, in fact, always in solved form[3] (e.g. $(\widehat{\alpha}_1 = A_1)$ and $(\widehat{\alpha}_2 = A_2)$ and $\dots$)? (We can think of this as a substitution, a rigid but relatively simple structure.) The "greedy" strategy does this: When we try to derive $A \leq \widehat{\alpha}$, or $\widehat{\alpha} \leq A$, we immediately choose A as the instance, conjoining $\widehat{\alpha} = A$ to the constraint (barely worthy of the name).

  But this approach is severely limited, as the following example (adapted from Pierce and Turner [PT98]) shows. Suppose

  $$f : \forall\alpha.\ \alpha \to \alpha \to \alpha,\ \ x : \mathsf{pos},\ \ y : \mathsf{std}$$

  and we need to instantiate α in the application $f\ x\ y$. (Perhaps f is a kind of choice operator, returning each argument with probability 1/2.) If we instantiate α to pos as suggested by the type of x, we cannot check $y \downarrow \mathsf{pos}$ since $\mathsf{std} \not\leq \mathsf{pos}$. If we try y's type, instantiating α to std, we will succeed since $\mathsf{pos} \leq \mathsf{std}$. This suggests that perhaps we just need some way of trying all the "obvious" instantiations of α. Unfortunately, if we instead had $\Gamma = \dots, x : \mathsf{even}$, $y : \mathsf{odd}$ neither x's nor y's type works since neither is a subtype of the other. In fact, α must be instantiated to a supertype of both types in question. In the first instance displayed above, one of these types ($\Gamma(y) = \mathsf{std}$) happened to be a supertype of the other, but some more general mechanism is needed.

  Union types are exactly that mechanism! Instantiating α to $\mathsf{even} \vee \mathsf{odd}$ works perfectly. But how does the typechecker come up with the union?

[3] We write $\widehat{\alpha}$ as the name of the existential induced by trying to instantiate α.

One answer is that the typechecker does nothing of the kind. The user must explicitly call for the union, not by wedging some sort of annotation inside the site of the polymorphic instantiation $f\ x\ y$ (which, to be consistent with our annotation scheme so far, should not need an annotation since it contains no checking forms in synthesizing position; see Remark 3.6). Instead, f's type must be changed:

$$f : \forall\alpha.\, \alpha \to \alpha \to \alpha \quad \Longrightarrow \quad \begin{array}{l} f : \forall\alpha_1.\, \forall\alpha_2.\, (\alpha_1 \vee \alpha_2) \to (\alpha_1 \vee \alpha_2) \to (\alpha_1 \vee \alpha_2) \\ \quad \textit{or} \\ f : \forall\alpha_1.\, \forall\alpha_2.\, \alpha_1 \to \alpha_2 \to (\alpha_1 \vee \alpha_2) \end{array}$$

The first rewrite corresponds to the original type with a union instantiating α; the second is more readable. For the first, the built-in backtracking of subtyping decomposes the union, choosing to derive either $\mathsf{even} \le \widehat{\alpha}_1$ or $\mathsf{even} \le \widehat{\alpha}_2$. If we try $\mathsf{even} \le \widehat{\alpha}_1$ first, we set $\widehat{\alpha}_1 = \mathsf{even}$. For the next argument, we have $\mathsf{odd} \le \widehat{\alpha}_1 \vee \widehat{\alpha}_2$. There, trying $\mathsf{odd} \le \widehat{\alpha}_1$ fails since (substituting for $\widehat{\alpha}_1$) $\mathsf{odd} \not\le \mathsf{even}$, but when we try $\mathsf{odd} \le \widehat{\alpha}_2$ we instantiate $\widehat{\alpha}_2 = \mathsf{odd}$, synthesizing $\widehat{\alpha}_1 \vee \widehat{\alpha}_2 = \mathsf{even} \vee \mathsf{odd}$ for the whole expression.

If that "manual" approach seems unsatisfying, consider that anything else shatters the subformula property: unions (and in other situations, intersections) are produced out of nowhere. The subformula property is not esoteric, but has immense practical value given the typechecking performance problems that unions and intersections sometimes cause, as in the bits-un.rml example (Listing 7.3 and Figure 6.5). Even if we improved our typechecking algorithms significantly, it is not at all clear that we could handle unions exploding out of every polymorphic instantiation.

Nonetheless, we have some ideas for "automatic" instantiation. The above example suggests the importance of the *number of occurrences* of α: since α appears twice in the domain of f, when f is applied two expressions will be checked against α, and it makes sense to instantiate α with a two-part union; if α appeared three times we might create a three-part union $\alpha_1 \vee \alpha_2 \vee \alpha_3$; if α appeared only once, no "split" would be necessary. Thus we could split only the nonlinear type variables. The position of α is also key: the αs are on the left hand side (negative position); if they were on the right, in the range of the function, an *intersection* split would be natural. (If α appeared more than once on each side, the left hand side should adequately constrain α and we would not introduce an intersection "lurking" on the right hand side.)

A more uniform approach would keep the union and intersection splits fluid; deriving $A_1 \le \widehat{\alpha}$ would *itself* introduce a new type variable and add $\widehat{\alpha} = A_1 \vee \widehat{\alpha}'$ to the constraint, keeping our options open for $A_2 \le \widehat{\alpha}$, which would become $A_2 \le A_1 \vee \widehat{\alpha}'$; the $\widehat{\alpha}'$ could be split in turn: $\widehat{\alpha}' = A_2 \vee \widehat{\alpha}'$, and so on, for however many $\ldots \le \widehat{\alpha}$ judgments we try to derive. We would need to get rid of the last existential variable introduced, perhaps by instantiating it to $\bot$. (After all, the subformula property is long gone.) If the existential appears on the left, $\widehat{\alpha} \le A_1$, introduce an intersection instead.

Finally, a historical note: the basic "greedy strategy" is, according to Pierce and Turner [PT98], the basis of one of Cardelli's System $F_{<:}$ typing algorithms [Car93]. Having no union types,

that algorithm cannot crack the std-pos example (which uses only atomic subtyping and should be within the purview of his system).

Our present inclination is to pursue the "manual" greedy approach, perhaps in combination with the directional modes suggested by Davies. If manual instantiation of unions and intersections turns out to be too cumbersome for users, we might pursue automatic instantiation methods, but such methods would probably have the simpler manual system as their foundation, making a manual system worth pursuing as a means if not an end.

Finally, we briefly consider a philosophical question about polymorphism. We presently have no parametric polymorphism, but we do have intersection polymorphism and index polymorphism (Π), and in our formulation these have an implicit character. That is, when a term of intersection type is used, the component is not indicated in the program: there is no marker saying which of $\wedge E_1$ and $\wedge E_2$ to apply. Likewise (and with the clearest analogy to parametric polymorphism) the index instantiating a Π type is not explicit in the term: for $x \uparrow \Pi a{:}\gamma.\, A_1 \to A_2$ the term is simply x, not some $x[i]$ containing the index i to substitute for a in $A_1 \to A_2$. In contrast, parametric polymorphism, while implicit at the level of source terms (the user applying the SML function *List.app* to a list of integers need not write *List.app*[int]), is typically made explicit during elaboration. Thus, as far as most of the compiler's phases are concerned, (parametric) polymorphism is explicit, and for good reason (if nothing else, one does not want to keep doing type inference to find the instantiations!).

One wonders, then, whether our implicit polymorphism is somehow inappropriate, due to being mismatched against tried-and-true explicit parametric polymorphism in SML compilers (among others). Our work so far is on type checking, not compilation. In a standalone SML typechecker that produced no intermediate code, the value of explicit parametric polymorphism would no longer be obvious, and might not even exist. So we see no particular reason to expect intersection and/or index polymorphism to be explicit in the setting of a standalone typechecker. In a compiler, the story is likely to be different. We have no clear idea of what a compiler based on intersection types, union types, etc.—in the same way that modern ML compilers are based on ordinary types in the sense of having typed intermediate languages, type-based optimizations, and so forth—would look like, but it could well call for explicit intersection and index polymorphism (recalling the classic compilation method of intersection types as pairs, unions as injections, etc.). In that case, there would no longer be any mismatch between parametric polymorphism and the forms we have explored in this thesis: all forms of polymorphism would be implicit at the source level, but made explicit during compilation.

8.1.2 Refinement-based compilation

Value-based refinements can make compiled code more efficient. For example, knowing that the subject of a case expression must be a particular constructor can eliminate a tag check. A special instance of this is when a Boolean expression is known to have type $\mathsf{bool}(\mathsf{true})$ or $\mathsf{bool}(\mathsf{false})$, as in **case** $x < y$ **of** $\mathsf{True} \Rightarrow e_1 \mid \mathsf{False} \Rightarrow e_2$ when $\Gamma = a{:}\mathcal{N}, b{:}\mathcal{N}, a < b, x{:}\mathsf{int}(a), y{:}\mathsf{int}(b)$: the test $x < y$ and the second case arm can be eliminated entirely. In fact, this corresponds to eliminating array bounds checks, an application of index refinements explored by Xi and Pfenning [XP98]. Unlike that system, ours has intersection and union types. Terms of such types might be compiled as pairs

and sums, at least in some cases, so that functions of intersection type would yield specialized code for each component of the intersection. Code specialization is nothing new, of course, but optimizations grounded in the user's type annotations could potentially surpass optimizations based on generic heuristics.

8.1.3 Index domains

While we have gone beyond the well-established domain of integers by supporting dimension refinements, further expansion is desirable. We outline a few candidate domains here.

- *Bit vectors*, not integers, are the appropriate index objects for "integer" arithmetic in Standard ML (and many other languages): the precision of `int` is limited and operations can overflow.[4] CVC Lite supports fixed-length bit vectors with several arithmetic operations, including standard binary addition.

- *Uninterpreted functions.* Index functions and predicates with no properties beyond congruence (if $i \doteq j$ then $f(i) \doteq f(j)$) are already supported by ICS and CVC Lite; there should be no difficulty in adding them.

- *Inductive families.* A form of ML-style datatype can be transposed into the index domain: each constructor c becomes an index function. Unlike uninterpreted functions, such functions should have inversion properties: $c(i) \doteq c(j)$ should imply $i \doteq j$. Xi [Xi00, pp. 13–16] gives an example of an evaluator for a tiny functional language, in which `typ` is an index sort representing types in the object language, `context` is an index sort representing deBruijn-style contexts, and object expressions are indexed by `typ` and `context` (the latter representing the object-language types of the expression's free variables):[5]

```
indexsort  typ = I | B | Fun of typ * typ
indexsort  context = CTXempty | CTXcons of typ * context
datatype  exp with typ * context
datacon One : -all t:typ, ctx:context- exp(t, CTXcons(t, ctx))
datacon Shift : -all ta,tb:typ- -all ctx:context-
                   exp(ta, ctx) → exp(ta, CTXcons(tb, ctx))
datacon Lam : -all ta,tb:typ- -all ctx:context-
                 exp(tb, CTXcons(ta, ctx)) → exp(Fun(ta, tb), ctx)
datacon App : -all ta,tb:typ- -all ctx:context-
                 exp(Fun(ta, tb), ctx) * exp(ta, ctx) → exp(tb, ctx)
...
```

  Neither ICS nor CVC Lite directly supports inductive families, though CVC Lite's successor CVC3 does [Bar06a]. In CVC Lite the inversion properties can be asserted as quantified

[4] Overflow in SML raises an exception, so the present indexing by integers should be sound: if a function of type $\Pi a{:}\mathcal{Z}.\, \mathsf{int}(a) \to \mathsf{int}(a+1)$ is applied to an integer n, any value actually returned will have value $n+1$; if overflow occurs, the function's result is effectively $\bot$, and $\bot \leq \mathsf{int}(n+1)$. In languages that ignore overflow, such as Objective Caml, we would have serious problems.

[5] We have adapted Xi's notation to make it more similar to ours.

formulas, but subsequent queries sometimes yield a result of "unknown due to quantifier instantiation".

- *Functional arrays.* Arrays (in ML terminology, vectors) with functional update $i[j := k]$ (denoting the array i with the jth element replaced by k) are supported by both ICS and CVC Lite [SBDL01]. We have not explored possible applications of this domain except in connection with sets, discussed next.

- *Uninterpreted sets with order.* In the red-black tree example in Chapter 7, we used refinements to capture the structural invariants that ensure red-black trees are balanced. We left out a more basic invariant: the set of keys/records stored. We would like to index red-black trees, and other container data structures, by that set. Order is another important property. An index sort ordset of ordered sets of otherwise unspecified objects would allow us to check the full specification of operations such as insertion. Plausible types for the Black constructor and *insert* function are as follows:

$$\begin{aligned}&\textbf{datacon}\ \mathsf{Black} : \Pi n{:}\mathcal{Z}.\, \Pi h{:}\mathcal{N}.\, \Pi L, R{:}\mathsf{ordset}.\\ &\qquad\qquad \big((l \in L) \Rightarrow (l < n)\big) \supset \big((r \in R) \Rightarrow (r > n)\big)\\ &\qquad\qquad \supset \big(\mathsf{int}(n) * \mathsf{dict}(h, L) * \mathsf{dict}(h, R) \to \mathsf{dict}(h+1, L \cup \{n\} \cup R)\big)\\ &\qquad \wedge \ldots\end{aligned}$$

$$(*[\ \ \textbf{val}\ \mathit{insert} : \Pi h{:}\mathcal{N}.\, \Pi S{:}\mathsf{ordset}.\, \mathsf{rbt}(h, S) * \mathsf{int}(n) \to \Sigma h'{:}\mathcal{Z}.\, \mathsf{rbt}(h', S \cup \{n\})\ \ ]*)$$

It should be possible to model ordering in the integer (or rational) domain; the additional operations and laws, such as $a + 0 = a$, would simply never come into play. The blurring of n as integer ($\mathsf{int}(n)$) and as member-of-a-set is somewhat curious—we might need a sort $\mathsf{ordered}$ of objects with an order, of which $\mathcal{Z}$ would be a "subsort". One hopes that adding such subsorting (not to be confused with datasort subsorting) would not have unfortunate consequences.

As for the set constraints, useful fragments of set theory are decidable, including:

  - MLSS (Multi-Level Syllogistic with Singleton), the unquantified theory with $\{\}$, $\cup$, $\cap$, $\setminus$, $=$, $\subseteq$, $\in$, and singletons $\{\text{—}\}$; see Cantone and Ferro [CF95];
  - MLSSF$^\forall$, which extends MLSS with uninterpreted functions and universal quantification (in a cumulative hierarchy); see Cantone and Zarba [CZ99].

MLSSF$^\forall$ is sufficient to write the types above, considering $<$ and $>$ to be uninterpreted functions.

As mentioned previously, ICS and CVC Lite support functional arrays. Couchot et al. [CDD+04] show how to reduce set-theoretic propositions to propositions in the theory of functional arrays mentioned previously. However, we have not determined if their method is in fact applicable in this setting; in particular, it is not clear if their method allows online assertion at reasonable cost.

Any decision procedure for sets must be usable in combination with other theories, e.g. in a Nelson-Oppen framework (see below). The particular case of sets is discussed by Kuncak and

Rinard [KR04]; the work of Cantone et al. [CF95, CZ99] may also be of value, since support for uninterpreted function symbols (from the perspective of sets, $<$, $>$, etc. are uninterpreted) is among the prerequisites for a Nelson-Oppen theory.

- *Regular languages.* For simple string processing, we might like to express that a given function's result contains its argument as a substring, that is, if $x = S_1$ then $f(x) = S_0 S_1 S_2$ for strings S_0, S_1, S_2. For example, a function to construct an error message that takes several arguments, one of which is an informational string, should include that string in the constructed message.

  This and many similar properties are within the scope of regular languages. The propositions P would include $s \in L$, where L is a language specified by a regular grammar or regular expression. We would need a procedure to decide relations such as $s \in L_1 \models s \in L_2$, which amounts to $L_1 \subseteq L_2$; fortunately, containment of regular languages is decidable [HU79].

  While it appears that no significant cooperation between a regular language constraint solver and the integer solver would be necessary (unless we extended the domain significantly, e.g. by allowing conversion between numeric strings and integers), we have not specified the domain precisely; doing so and writing the appropriate solver could be nontrivial.

- *Context-free languages.* For properties of more complex string processing, such as fully verifying the correctness of (say) recursive descent parsers, regular languages are not powerful enough. Following the approach to regular languages just discussed, we must decide $L_1 \subseteq L_2$ where L_1 and L_2 are context-free languages. Unfortunately, this is undecidable [Hop69]. However, an incomplete but sound algorithm might suffice; this is the approach followed by Thiemann [Thi05], whose system supports refinement of strings by context-free languages.

We note that while constraint solvers (perhaps more properly, decision procedures) are a hot area of research (and probably will remain so), our application seems to be atypical. Speed is much less important in our setting than in many others; unlike say circuit verification, our constraints are quite small. While there is no problem *in principle* with using a system that works well on huge problems to solve tiny ones, in practice this consideration may conflict with another unusual requirement: correctness. Unlike many "bug-finding" approaches to software quality, our approach means that we can at least *hope* that when our typechecker says a program is type-correct it actually is—not just that no specific bug has been found. In bug-finding, a constraint solver that is occasionally wrong is tolerable, whereas one that would take decades to give an answer is utterly useless. Their engineering requirements therefore emphasize speed over correctness. But in our setting, our constraints are so small that naïve algorithms may be good enough, yet correctness is paramount.[6]

An important issue is *combination* of index domains. We are fortunate that dimension equalities can be reduced to sets of integer equalities, which makes it easy to support integers and dimensions simultaneously. In general, to combine theories one needs a generic *combination procedure*. Both

[6] As a depressing aside, SRI recently (2006) abandoned ICS in favor of a new system called Yices. Unlike ICS, Yices lacks persistent state, so we would have to "replay" assertions as we do for CVC Lite. Also unlike ICS, Yices is implemented in C++ rather than OCaml. In the interests of decorum, we decline to give our thoughts on whether persistence and implementation language are related.

ICS and CVC Lite use such machinery internally, but neither provides a good interface for plugging in solvers for new domains[7]. On a positive note, the theoretically inefficient but elegant Nelson-Oppen combination procedure [NO79] might be enough for our purposes, allowing us to avoid Shostak's method [Sho84, RS01, SR02].[8]

We hope that there are other applications with similar requirements, and that more systems of the kind we need will be built, possibly through an open-ended interface for plugging in simple constraint solvers (perhaps based on rewriting systems [Mit96, BN98]).

8.1.4 Mutable references

Neither our formal systems nor our implementation have SML's **ref**, nor any other form of mutable storage. Adding **ref** to the implementation should be straightforward, but awkward: without parametric polymorphism we would need a family of types A **ref** and would probably have to add to lib_basis.rml lines such as

```
primitive fun ! : (int ref → int) & (bool ref → bool) & ...
```

Extending the formal systems is a weightier matter. However, we believe that our property type rules $\wedge$I, $\vee$E, and so on would be sound without modification (except the straightforward addition of store typings). Our $\wedge$I is the same as that of Davies and Pfenning [DP00], which they proved sound in the presence of **ref**. In particular, in their proof of safety, restricting $\wedge$I to values ensured consistency of the store typings obtained by applying the IH to each premise. That argument should hold for our system's $\wedge$I. In $\vee$E, we type the same term twice, but we do not apply the IH to both of those premises, since value definiteness allows us to conclude either $v : A$ or $v : B$ from $v : A \vee B$ and thereby focus on a single premise, on which the IH would yield a single store typing.

8.1.5 Call-by-name languages

We have worked in a call-by-value semantics throughout the thesis. What about other formulations such as call-by-name and call-by-need [AFM$^+$95] ("lazy evaluation")? In this section we sketch some ideas for how to adapt our work to call-by-name (cbn)[9]. To keep things simple, we strip down the system so the only types are $\mathbf{1}$, $\rightarrow$, $\wedge$, $\vee$, $\top$ and $\bot$ and the only terms are $()$, x, $\lambda x.\, e$, $e_1\, e_2$, u and **fix** $u.\, e$. Even this suffices to look into key issues surrounding union types.

Does anything need to change? After all, type safety holds for the simply typed lambda calculus under either reduction strategy. This also appears to be the case in our setting; unfortunately, the resulting type system is crippled under cbn. We start by explaining why, and then discuss a few ideas for alternative formulations of $\vee$E and related rules.

[7] To "protect" SRI's "intellectual property", the ICS source code is unavailable; the successor project, Yices, is similarly "protected". CVC Lite is open but written in C++. Implementation language issues are not insurmountable, but life is short.

[8] It is not entirely unfair to sum up the difference between the Nelson-Oppen and Shostak algorithms by observing that some time after the respective seminal papers, there appeared a new proof of correctness of the former [TH96] and a *disproof* of the correctness of the latter [RS01].

[9] The interplay between intersection/union types and reduction strategies is presently being studied, with a theoretical emphasis, by Zeilberger, as part of what he calls *operationally-sensitive typing phenomena* [Zei07].

Note that we specifically assume a pure deterministic semantics. (Our cbv semantics happens to be deterministic, but that is not essential to our results.)

Straightforward adaptation to cbn

Recall the union elimination rule from Chapter 2.

$$\frac{\Gamma \vdash e' : A \vee B \qquad \Gamma, x{:}A \vdash \mathcal{E}[x] : C \qquad \Gamma, y{:}B \vdash \mathcal{E}[y] : C}{\Gamma \vdash \mathcal{E}[e'] : C} \vee\mathrm{E}$$

This rule is a good representative of the family of rules $\{\bot\mathrm{E}, \vee\mathrm{E}, \Sigma\mathrm{E}, ⅋\mathrm{E}, \mathsf{direct}\}$. The rule requires that the subterm e' of union type be in evaluation position. Under cbn, function arguments are not evaluated, so $v\mathcal{E}$ is not an evaluation context:

$$\begin{array}{ll} [\quad \mathcal{E} ::= [\,] \mid \mathcal{E}e \mid v\mathcal{E} \quad] & \text{call by value} \\ \quad\;\; \mathcal{E} ::= [\,] \mid \mathcal{E}e & \text{call by name} \end{array}$$

We believe rule $\vee$E is sound under both cbv and cbn with this new definition of $\mathcal{E}$s. Under cbv, $\mathcal{E}[e']$ takes a step if either e' does or e' is a value; under cbn the latter case is excluded—$\mathcal{E}[e']$ takes a step only if e' does. In the proof case for type safety under cbn, it appears one would simply consider the two possibilities: e' takes a step, in which case the logic is the same as the cbv case, or e' is a value, in which case we have a value (e') to add to σ, allowing us to apply the IH.

However, $\vee$E with the cbn version of $\mathcal{E}$ is too restrictive. For example, we cannot now use $\vee$E to derive

$$f : (A \to C) \wedge (B \to C), x : A \vee B \vdash f\ x : C$$

since x is not in evaluation position. A term such as $(f\ y)\ z$ where z is of union type is likewise inadmissible; we can apply direct to get $\bar{\mathbf{x}}\ z$ but z is not in evaluation position in cbn.

A value restriction?

A value restriction on e' in $\vee$E [vBDCdM00] is a non-starter: for type preservation under β-reduction, one needs a generalized substitution property that allows even non-values to be substituted for x, but that generalization does not hold for value-restricted rules (whether a modified $\vee$E or $\wedge$I/$\supset$I/$\top$I): consider $\wedge$I applied to x, a value; after substituting a non-value for x we can no longer apply $\wedge$I. It seems that for $\wedge$I and $\supset$I we can simply remove the value restriction, which was motivated by soundness under effects (such as mutable references); what to do with $\top$I is less clear. In any case, it appears that we cannot help matters by putting a value restriction on the subterm e' of $\vee$E's subject.

Strictness restriction

If we restrict e' to be in some *strict position* $\mathcal{S}[e']$, union-elimination should be safe. Formulating such positions would be nontrivial, but might be the only way to obtain a useful rule.

Call-by-need vs. call-by-name

Working in an explicitly call-by-need system might allow a value restriction to be imposed: instead of $(\lambda x.\, e)e' \mapsto [e'/x]e$ we would have $(\lambda x.\, e)e' \mapsto \textbf{let}\, x = e' \,\textbf{in}\, e$. This seems worth investigating, especially since call-by-name systems are typically implemented via call-by-need anyway. Moreover, in the case where $\vee$E is applied with $e' = x$, call-by-need suspiciously resembles the parallel reduction semantics studied by others.

8.1.6 Evidence of things unseen

Our work expands the capabilities of static type systems by making more properties expressible. Yet there will always be properties that, even with guidance in the form of type annotations, cannot be expressed without losing decidability. Furthermore, components such as object-code libraries are simply impervious to typechecking. We suggest several ways to mitigate such difficulties.

The first is a fallback strategy of *runtime refinement checking*. For example, if we claim that some library function f has type

$$f : \Pi a{:}\mathcal{N}.\, \mathsf{int}(a) \to \mathsf{list}(a)$$

we could wrap every call to f with a check that the length of the resulting list is equal to its argument. One mechanism for this would be a *soft annotation*. The idea is that we erase such an annotation if the typechecker can determine whether or not the annotated code checks against the type; if it cannot, because the code is in an external library, we generate a wrapper function that includes a call to a (perhaps automatically generated) checking function.

As the above example illustrates, such checks could take time linear (at least!) in the size of the data structures involved. Moreover, if the data is functional, we would need to build a new copy of the data with the functions replaced by wrapped functions. This suggests a use for a third kind of annotation, a *sharp annotation*, which is *not even checked at runtime*. As the name suggests, this is perilous: if the property claimed by the sharp annotation does *not* hold, the typechecker will happily reason from that false assumption, drawing all manner of wrong conclusions.

A more compelling need for sharp annotations arises out of the index refinements. Some constraint domains involve properties that cannot be checked at runtime. A simple example is found in the seemingly innocuous theory of order, where the index objects have an operation $\preceq$ which must be reflexive, antisymmetric and transitive. If we use this theory with a primitive type such as the integers and assume the built-in integer comparison function has the necessary properties, all is well, since that function really does have those properties. But if we need the theory of order for some user-defined type, such as keys in a database with various components lexicographically ordered, the comparison function *cmp* comes from the user, so we would like to check that *cmp* is reflexive, antisymmetric and transitive. However, the property of transitivity is neither expressible in the refinement system nor checkable at runtime! In this case, the form of the sharp annotation is different:

$$\mathit{cmp} :\sharp: \textsc{transitive}$$

The constraint solver furnishes the information that TRANSITIVE denotes a property. There can be no check that *cmp* actually is transitive, but the programmer must explicitly claim that the property holds.

We discussed external libraries because they are obviously impervious to static typechecking. But programs can grow so large that parts of the program may as well be an external library. Suppose there is a module M with datatype τ which M's programmer refined as they saw fit; years later someone working on another part of the program needs an unforeseen refinement of τ, but the benefit from static checking may not justify the effort involved in modifying M. In such a situation, soft or sharp annotations would be expedient.

Ideally, the system would be able to use evidence of various kinds to support sharp annotations. A key goal of the Programatica project [Pro03] is to build a system that tracks various kinds of evidence, from claims of the kind just discussed to proofs produced by full-blown theorem provers. Integrating type refinements with a Programatica-like system could be quite powerful. Other approaches, which we briefly examined in Section 1.6.3, aim to combine automatic checking of some properties with user proofs of others; these could play a very useful role.

The Extended Static Checker, which we discussed in Section 1.7.2, has **assume** statements roughly corresponding to our "sharp annotations", and Leino [Lei01] suggests falling back on dynamic checks (our "soft annotations") or **assume** statements in cases where static checking is undecidable.

8.1.7 Derivation generation

The typechecker could be extended to generate a derivation if typechecking succeeds. A standalone program to check a derivation's validity would be far simpler and (in general) much faster than generating the derivation through typechecking, since there would be no backtracking.

A key benefit would be to increase confidence in the typechecker: the pronouncement "Program typechecks" would be given much more weight, especially if derivation generation were combined with a proof-generating decision procedure for the index domain(s).

Saving the generated derivations could enable some form of incremental typechecking by allowing the typechecker to refer to a previously saved derivation as a first guess for nondeterministic choices: if the saved derivation moved from $x{:}A_1 \wedge A_2$ to $x{:}A_2$ while checking a use of x, the choice $x{:}A_2$ could be tried first, even though the system would normally try it second. Obviously, such a scheme's effectiveness depends on how resilient derivations are in the face of small changes to source code. (A variant of the scheme would store only an oracle string and use that to decide which choices to try first.)

8.1.8 Counterexample generation

When a program fails to typecheck, it would be useful to know why. An intuitively appealing idea is to generate a counterexample. For instance, if a function does not check against $\mathsf{red} \rightarrow \mathsf{black}$, give the user a value $v : \mathsf{red}$ such that the result of the function is not black. However, this is not a trivial problem, especially with higher-order functions, as we then need to generate functions as counterexamples. Finally, the technique is not directly applicable to invaluable refinements such as dimensions: there is no value of simple type real that is not also of type $\mathsf{real}(d)$ for any dimension d!

8.1.9 Suggestion tools for refinements

Our experience with red-black tree deletion (Section 7.2.5) indicates that coming up with appropriate refinements and type annotations for legacy code, while productive in terms of finding bugs, is not easy. For value refinements, it could be useful to apply tools analogous to Daikon [ECGN01] to find alleged invariants, from which the user could—if convinced that the invariant *should* hold and is not evidence of a bug—derive a possible type annotation.

Appendix A

Guide to Notation

Roman

a, b	index variables	
c	datatype constructors	
e	expressions (terms, program terms)	
$\check{e}$	pre-values	Chapter 5, p. 121
$\widehat{e}$	anti-values	Chapter 5, p. 121
i, j, k	index-domain expressions; also subscripts (e.g. A_i)	
ms	matches (case arms)	
p	patterns	Chapter 4, p. 100
s	states in Solve	Chapter 6
u	fixed point variables	
v	values	
other lowercase	program variables, index variables, meta-variables	
A, B, C, D	types	p. 32
As, Bs	contextual typing annotations	Chapter 3
BLV	bound linear variables	
FLV	free linear variables	
FV	free (program, index) variables	
L	lists of bindings	Chapter 5
$N^d_{\mathbf{R}}$	floating-point value N of dimension d	Chapter 7
P	index-domain propositions	

Bold

$\bar{\mathbf{a}}, \ldots, \bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}$	linear variables	Chapters 3 and 5

Greek

γ	index sorts	
δ	datasorts	
ϵ	empty bitstring	
μ	measure for induction	Def. 5.77, p. 160
ρ	renaming (variable-for-variable substitution)	Chapter 3
σ	substitution (for index variables and/or program variables)	
Γ	contexts (environments) for program variable typings, index variable typings, and propositions	
Δ	contexts (environments) for linear variables	
Θ	operators (in pattern checking)	Figure 4.3
Π	universal index quantifier	
Σ	existential index quantifier	
Ω	external solver-level context	Chapter 6

Script

$\mathcal{C}$	syntactic contexts, unrestricted	
$\mathcal{D}$	derivations	
$\mathcal{D} :: \ldots$	$\mathcal{D}$ derives judgment "…"	
$\mathcal{E}$	syntactic contexts, evaluation	
$\mathcal{J}$	judgments	Chapter 4
$\mathcal{Q}$	syntactic contexts, "elongated evaluation"	Figure 5.1
$\mathcal{R}$	rules	e.g. Def. 2.15
$\mathcal{W}$	let-free viable paths	Def. 5.32, p. 135

Angular/ Rounded

$\leq$	is a subtype of	Figure 2.11
$\lesssim$	is a contextual subtype of	Section 3.4.3
$\bot$	falsehood (index-level proposition)	
$\bot$	empty type	
$\top$	top (greatest) type	
$\wedge$	intersection type	
$\wedge$	conjunction (index-level proposition)	Chapter 6
$\vee$	union type	
$\vee$	disjunction (index-level proposition)	Section 6.9
$\supset$	guarded type	Chapter 2
$\between$	asserting type	Chapter 2

Arrows

$\mapsto$	steps to	Chapter 2
$\mapsto_R$	reduces to	Chapter 2
$\mapsto_M$	reduces to (“tiny-step” pattern matching)	Chapter 4
$\Rightarrow$	implication (index-level proposition)	
$\uparrow$	synthesizes	(“in the present system”)
$\downarrow$	checks against	(“in the present system”)
$\uparrow^{\mathsf{tri}}$	synthesizes (tridirectional system)	Chapter 3
$\downarrow^{\mathsf{tri}}$	checks against (tridirectional system)	Chapter 3
$\uparrow^{\mathbb{L}}$	synthesizes (left tridirectional system)	Chapters 3 and 5
$\downarrow^{\mathbb{L}}$	checks against (left tridirectional system)	Chapters 3 and 5
$\uparrow^{\mathsf{let}}$	synthesizes (let-normal system)	Chapter 5
$\downarrow^{\mathsf{let}}$	checks against (let-normal system)	Chapter 5
$\downarrow\uparrow$	checks against *or* synthesizes	
$\underline{\downarrow\uparrow} \ldots \underline{\downarrow\uparrow}$	two (or more) judgments having the same direction: checks against ... checks against, *or* synthesizes ... synthesizes	

Other

$\|e\|$	erasure of typing annotations	Chapters 3 and 4
$\|e\|$	Xi's let-normal translation [Xi98]	Chapter 5
☞	points to part of what is to be shown	Section 1.9.2

Extended BNF (Chapter 6)

$[\ldots]$	zero or one occurrences of '...'
$(\ldots)^*$	zero or more occurrences of '...'
$(\ldots)^+$	one or more occurrences of '...'

Sources of Quotations

p. 5 Mukesh Agrawal.
Personal communication.

p. 9 Jean-Yves Girard.
Bulletin of Symbolic Logic, 2003, p. 140.

p. 223 G. Baldwin.
SIGPLAN Notices, August 1987.

p. 233 G. Baldwin.
SIGPLAN Notices, August 1987.

p. 233 Edsger W. Dijkstra.
EWD 1243, http://www.cs.utexas.edu/users/EWD/.

Bibliography

[AC98] Roberto Amadio and Pierre-Louis Curien. *Domains and lambda calculi*, volume 46 of *Cambridge Tracts in Theoretical Comp. Sci.*, chapter 3. Cambridge Univ. Press, 1998. 1.6.1, 3.1

[ACL+04] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-oriented units of measurement. *SIGPLAN Notices*, 39(10):384–403, 2004. Originally presented at OOPSLA '04. 7.4, 7.4.6

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. In *ACM Symp. Principles of Programming Languages (POPL '95)*, pages 233–246, 1995. 8.1.5

[AH05] David Aspinall and Martin Hofmann. Dependent types. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 2, pages 46–86. MIT Press, 2005. 1.6.3

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic and Computation*, 2(3):297–347, 1992. 6.7.1

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *Int'l Conf. Functional Programming (ICFP '98)*, pages 239–250, 1998. 1.6.3

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symp. Principles of Programming Languages (POPL '94)*, pages 163–173, 1994. 1.6.1

[Bar06a] Clark Barrett. CVC 3.0 website. `http://www.cs.nyu.edu/acsys/cvc3/`, 2006. 6.5.2, 8.1.3

[Bar06b] Clark Barrett. CVC Lite website. `http://www.cs.nyu.edu/acsys/cvcl/`, 2006. 6.5.2

[BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In *Int'l Conf. Computer Aided Verification (CAV '04)*, volume 3114 of *LNCS*, pages 515–518. Springer, July 2004. 6.5.2

[BCKW05] Adam Bakewell, Sébastien Carlier, A. J. Kfoury, and J. B. Wells. Inferring intersection typings that are equivalent to call-by-name and call-by-value evaluations. Technical report, Church Project, Boston University, April 2005. 1.6.1

[BDCd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119:202–230, 1995. 2.1, 2.4.1, 2.8, 3.5

[BDF+04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. 1.7.2

[BDL96] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In *Int'l Conf. Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 187–201. Springer, November 1996. 6.5.2

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004. 1.7.2

[Blu01] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001. 7.4.7

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. 8.1.3

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. 1.6.3

[Cam] Caml Light website. `http://caml.inria.fr/caml-light/`. 1.6.3

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Research report 97, DEC/Compaq Systems Research Center, February 1993. 8.1.1

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. Principles of Programming Languages (POPL '77)*, pages 238–252, 1977. 1.6.1

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981. 1.6.1, 3.1

[CDD+04] Jean-François Couchot, Frédéric Dadeau, David Déharbe, Alain Giorgetti, and Silvio Ranise. Proving and debugging set-based specifications. *Electronic Notes in Theoretical Computer Science*, 95:189–208, 2004. 8.1.3

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`, 1997. Release of 1 October 2002. 1.1

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991. 1.6.1

[CF95] D. Cantone and A. Ferro. Techniques of computable set theory with applications to proof verification. *Comm. Pure and Applied Math.*, 48:901–945, 1995. 8.1.3

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988. 1.6.3

[CH03] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003. 7.4.7

[Chr98] Jacek Chrząszcz. Polymorphic subtyping without distributivity. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 346–355. Springer, 1998. 8.1.1

[CKP05] Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. *J. Functional Programming*, 15(2):249 – 291, 2005. 12

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. 7.2.4, 7.2.5, 7.2.5, 7.2.5

[CW04] Sébastien Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. In *Workshop on Intersection Types and Related Systems (ITRS '04)*, pages 173–202, 2004. 1.6.1

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Int'l Conf. Functional Programming (ICFP '05)*, pages 66–77, 2005. 1.6.3

[CZ99] Domenico Cantone and Calogero G. Zarba. A tableau-based decision procedure for a fragment of set theory involving a restricted form of quantification. In *Int'l Conf. Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX '99)*, LNAI, pages 97–112. Springer, 1999. 8.1.3

[Dav97] Rowan Davies. A practical refinement-type checker for Standard ML. In *Algebraic Methodology and Software Tech. (AMAST'97)*, pages 565–566. Springer LNCS 1349, 1997. 1.6.1

[Dav05a] Rowan Davies. *Practical refinement-type checking*. PhD thesis, Carnegie Mellon University, 2005. CMU-CS-05-110. 1.1, 1.3, 1.6.1, 1.6.1, 2, 2.3.2, 2, 3.1, 3.3, 3.4.1, 4.3, 4.9.2, 6.1, 6.7.2, 6.10, 6.12, 6.13, 7.2.3, 7.2.4, 8.1.1

[Dav05b] Rowan Davies. SML CIDRE distribution page. `http://www.cs.cmu.edu/~rowan/sorts.html`, 2005. 1.6.1

[DD01] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: on the impact of the CPS transformation. Technical Report BRICS-RS-01-54, University of Aarhus, December 2001. 5.9

[DE73] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *J. Combinatorial Theory (A)*, 14:288–297, 1973. 7.2.2

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-159, Compaq SRC, 1998. 1.7.2

[dMOR+04] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and Natarajan Shankar. The ICS decision procedures for embedded deduction. In *Int'l Joint Conf. Automated Reasoning (IJCAR '04)*, volume 3097 of *LNCS*, pages 218–222, Cork, Ireland, June 2004. 6.5.1

[DP00] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000. 1.6.1, 2.1, 2.3.2, 2.7, 2.9, 3.1, 3.3.1, 7.2.3, 8.1.4

[DP03] Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FOSSACS '03)*, pages 250–266, Warsaw, Poland, April 2003. Springer LNCS 2620. 1

[DP04a] Jana Dunfield and Frank Pfenning. Tridirectional typechecking. Technical Report CMU-CS-04-117, Carnegie Mellon University, March 2004. Extended version of [DP04b]. 1

[DP04b] Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In X. Leroy, editor, *ACM Symp. Principles of Programming Languages (POPL '04)*, pages 281–292, Venice, Italy, January 2004. A

[Dun02] Jana Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002. 2.1, 2.3.4

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Soft. Eng.*, 27(2):99–123, February 2001. 8.1.9

[EJC01] Edward A. Euler, Steven D. Jolly, and H. H. 'Lad' Curtis. The failures of the Mars Climate Orbiter and Mars Polar Lander: a perspective from the people involved. In *24th Annual AAS Guidance and Control Conf.* American Astronautical Society, 2001. `http://brain.cs.uiuc.edu/integration/AAS01_MCO_MPL_final.pdf`; also appears in Advances in the Astronautical Sciences, volume 107, pages 635–656. 7.4.6

[Els05] Martin Elsman et al. ML Kit website. `http://www.itu.dk/research/mlkit/`, 2005. 1.6.1

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Int'l Conf. on Functional Programming (ICFP'02)*, pages 48–59, October 2002. 1.7.1

[FLMP99] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Int'l Conf. Functional Programming (ICFP '99)*, pages 114–125, 1999. 7.4.7

[Fos02] Jeffrey Scott Foster. *Type qualifiers: lightweight specifications to improve software quality*. PhD thesis, University of California, Berkeley, 2002. 7.4.7

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, pages 268–277. ACM Press, 1991. 1.3, 1.6.1, 3.3

[FP06] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. ArXiv postprint, `http://arxiv.org/abs/cs.PL/0403034`, January 2006. 7.4.7

[Fre94] Tim Freeman. *Refinement types for ML*. PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-110. 1.1, 3.3

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI '93)*, pages 237–247, 1993. 5.3

[Gir03] Jean-Yves Girard. From foundations to ludics. *Bulletin of Symbolic Logic*, 9(2):131–168, 2003. 1.6.1

[GR04] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004. 7.2

[GS78] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science*, pages 8–21, 1978. 7.2.4

[Hay94] Susumu Hayashi. Singleton, union, and intersection types for program extraction. *Information and Computation*, 109:174–210, 1994. 1.6.1

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. 1.6.3

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969. 5.3.1

[Hop69] John E. Hopcroft. On the equivalence and containment problems for context-free languages. *Math. Systems Theory*, 3(2):119–124, 1969. 8.1.3

[Hou83] R. T. House. A proposal for an extended form of type checking of expressions. *Computer Journal*, 26(4):366–374, 1983. 7.4

[HP99] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999. 3.6.2

[HS97] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. Technical Report CMU-CS-97-147, Carnegie Mellon University, 1997. A shorter version appeared in Proof, Language and Interaction: Essays in Honour of Robin Milner. 4.9.1

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. 1.1, 8.1.3

[IN06] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Symp. Applied Computing (SAC '06)*, pages 1435–1441, 2006. 2.8

[Jim95] Trevor Jim. What are principal typings and what are they good for? Technical memorandum MIT/LCS/TM-532, MIT, November 1995. 3.1, 3.6.3

[Kah01] Stefan Kahrs. Red-black trees with types. *J. Functional Programming*, 11(4):425–432, July 2001. 7.2.4

[Ken96] Andrew Kennedy. *Programming languages and dimensions*. PhD thesis, University of Cambridge, 1996. TR No. 391, University of Cambridge Computer Laboratory. 7.4, 7.4.2, 7.4.4, 7.4.5, 7.4.5, 7.4.6

[Kfo00] Assaf J. Kfoury. A linearization of the lambda-calculus. *J. Logic Comput.*, 10(3), 2000. 3

[KMM91] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 444–478. 1991. 1.6.1

[KR04] Viktor Kuncak and Martin Rinard. On decision procedures for set-valued fields. Technical Report 975, MIT CSAIL, November 2004. 8.1.3

[KTU94] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2):368–398, 1994. 1.6.1

[KW04] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1–3):1–70, 2004. 1.6.1

[Lan51] Henry L. Langhaar. *Dimensional Analysis and Theory of Models*. Wiley, 1951. 7.4.5

[Lei01] K. Rustan M. Leino. Extended Static Checking: A ten-year perspective. In *Dagstuhl Anniversary Conf.*, volume 2000, pages 157–175, 2001. 1.7.2, 8.1.6

[LH05] Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University, 2005. 1.6.3

[Lit03] Vassily Litvinov. *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington, 2003. 2.8

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *USENIX Conf. Domain-Specific Languages (DSL '99)*, pages 109–122, October 1999. Also appeared in ACM SIGPLAN Notices 35 (1), Jan. 2000. 7.4.7

[Log87] J. David Logan. *Applied Mathematics: A Contemporary Approach*. Wiley, 1987. 7, 7.4.5

[Mic68] Donald Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, 1968. 6.7.2

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17(3):348–375, 1978. 1

[Mit96] John C. Mitchell. *Foundations for Programming Languages*, pages 203–233. MIT Press, 1996. 8.1.3

[MM04a] Conor McBride and James McKinna. The view from the left. *J. Functional Programming*, 14(1):69–111, 2004. 1.6.3

[MM04b] Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *Int'l Conf. Functional Programming (ICFP '04)*, pages 138–149, Snowbird, Utah, USA, September 2004. 1.6.1, 1.6.1

[Mog88] Eugenio Moggi. Computational lambda-calculus and monads. Technical Report ECS-LFCS-88-66, University of Edinburgh, 1988. 2.3.2, 5.3

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986. 2.8

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 2.1, 4.9.1, 6, 6.1.6

[MW03] Kenneth MacKenzie and Nicholas Wolverson. Camelot and Grail: resource-aware functional programming on the JVM. `http://groups.inf.ed.ac.uk/mrg/publications/mrg/`, 2003. 5.3

[MWH03] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Int'l Conf. Functional Programming (ICFP '03)*, pages 213–226, Uppsala, Sweden, September 2003. 7.4.7

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symp. Principles of Programming Languages (POPL '99)*, pages 228–241, 1999. 12

[NGd94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1994. 1.6.3

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Prog. Lang. Sys.*, 1(2):245–257, 1979. 6.2, 8.1.3

[Obe99] James Oberg. Why the Mars probe went off course. *IEEE Spectrum*, 36(12), December 1999. `http://www.jamesoberg.com/mars/loss.html`. 7.4.6

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge, 1998. 7.2.5

[Pie91a] Benjamin C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-205. 1.6.1, 3.1, 3.4.1

[Pie91b] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991. 2.1, 6, 2.8, 3

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994. Summary in *ACM Symp. Principles of Programming Languages (POPL '93)*. 8.1.1

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. 1.9

[PP01] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Functional Programming*, 11(3):263–317, 2001. 1.6.1

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksells, 1965. 1.9, 2.2, 2.3.2, 2.4.1, 3.2, 6.11.2

[Pro03] Programatica project website. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`, 2003. 8.1.6

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *ACM Symp. Principles of Programming Languages*, pages 252–265, 1998. Full version in *ACM Trans. Prog. Lang. Sys.*, 22(1):1–44, 2000. 1.4, 3.1, 3.6.2, 6.12, 8.1.1

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE Conf. Supercomputing*, pages 4–13, 1991. 7.2.2

[PVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Int'l Conf. Functional Programming (ICFP '06)*, Portland, Oregon, September 2006. 7.4.7, 13

[PW03] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. *J. Functional Programming*, 13(5):905–923, 2003. 5.9

[Rep01] John Reppy. Local CPS conversion in a direct-style compiler. In *ACM Workshop on Continuations (CW '01)*, pages 13–22, 2001. 5.3

[Rey67] John C. Reynolds. Automatic computation of data set definitions. `ftp://ftp.cs.cmu.edu/user/jcr/autodataset.pdf`. A shorter version appears in *Information Processing 68*, November 1967. 1.6.1

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, 1983. `http://www.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf`. 7.4.5

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Report CMU-CS-88-159, Carnegie Mellon University, 1988. Superseded by [Rey96]. 1.6.1, 3.4.1, A

[Rey93] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3–4):233–247, 1993. 5.3

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996. Supersedes [Rey88]. 1.6.1, 1.6.1, 3.4.1, 6.10, A

[RP96] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In *European Symp. Programming (ESOP '96)*, pages 296–310, Linköping, Sweden, April 1996. Springer LNCS 1058. 3.2

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *Logic in Computer Science (LICS '01)*, pages 19–28, 2001. 8.1.3, 8

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *Int'l Conf. Computer Aided Verification (CAV '02)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002. 6.5.2

[SBDL01] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS '01)*, pages 29–37, 2001. 8.1.3

[SF94] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI '94)*, pages 1–12, June 1994. 5.9

[She04] Tim Sheard. Languages of the future. *SIGPLAN Notices*, 39(12):119–132, December 2004. Originally presented at OOPSLA '04. 1.6.3, 7.4.7

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984. 8.1.3

[SP04] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *4th Int'l Workshop on Logical Frameworks and Meta-languages (LFM '04)*, pages 106–124, July 2004. 1.6.3, 7.4.7

[SR02] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *Int'l Conf. Rewriting Techniques and Applications (RTA '02)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002. 8.1.3

[SRI03] SRI International. ICS website. `http://www.icansolve.com`, 2003. 6.5.1

[TH96] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Int'l Workshop on Frontiers of Combining Systems*, Applied Logic, pages 103–120, March 1996. 8

[Thi05] Peter Thiemann. Grammar-based analysis of string expressions. In *Workshop on Types in Language Design and Impl. (TLDI '05)*, pages 59–70, Long Beach, Calif., 2005. 8.1.3

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *SIGPLAN Conf. Programming Language Design and Implementation (PLDI '96)*, pages 181–192, 1996. 5.3

[TO98] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *J. Functional Programming*, 8(4):367–412, 1998. 5.3, 6.13

[vBDCdM00] Steffen van Bakel, Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Yoko Motohoma. The minimal relevant logic and the call-by-value lambda calculus. Technical Report TR-ARP-05-2000, The Australian National University, August 2000. 2.4.1, 2.8, 8.1.5

[Wad89] Philip Wadler. Theorems for free! In *Symp. Functional Programming Languages and Computer Architecture (FPCA) '89*, pages 347–359, September 1989. 7.4.5

[WDMT02] J.B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Functional Programming*, 12(3):183–317, May 2002. 1.6.1

[Wel02] J.B. Wells. The essence of principal typings. In *Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002. 3.1, 3.6.3

[WH02] J. B. Wells and Christian Haack. Branching types. In *European Symposium on Programming (ESOP '02)*, pages 115–132, 2002. 3.1

[WO91] Mitchell Wand and Patrick M. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 479–483. 1991. 7.4

[WSW05] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *Int'l Conf. Functional Programming (ICFP '05)*, pages 268–279, 2005. 1.6.3

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symp. Principles of Programming Languages (POPL '03)*, pages 224–235, 2003. 7.4.7

[Xi98] Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998. 1, 1.2, 1.6.3, 2.1, 2.3.4, 2.3.5, 2.4, 2.4.5, 2.8, 3.2, 3.4.2, 3.6.1, 5.2.1, 6.7.2, 6.12, 7.1, 7.2.4, 8.1.1, A

[Xi00] Hongwei Xi. Dependently typed data structures. Revision superseding WAAAPL '99; `http://www.cs.bu.edu/~hwxi/academic/papers/DTDS.pdf`, February 2000. 2.3.4, 2.3.5, 8.1.3

[Xi01] Hongwei Xi. Dependent types for program termination verification. In *Logic in Computer Science (LICS '01)*, pages 231–242, June 2001. 1.6.3

[Xi02] Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, October 2002. 1.6.3

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *TYPES 2003*, LNCS, pages 394–408. Springer, 2004. 2.3.5

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI '98)*, pages 249–257, 1998. 8.1.2

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symp. Principles of Programming Languages (POPL '99)*, pages 214–227, 1999. 1.3, 2.1, 2.3.4, 2.4, 2.8, 3.6.1

[Zda02] Stephan A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002. 12

[Zei07] Noam Zeilberger. The logical basis of evaluation order. PhD thesis proposal, Carnegie Mellon University, 2007. `http://www.cs.cmu.edu/~noam/research/proposal.pdf`. 9

Index

:!, 177
Ω, 181
μ, 37
Rank, 37
Size, 37
↦, 20
A^{con}, 24
Ωmega, 11
☞, 15
lib_basis.rml, 173
0-CFA, 166

A-normal form, 118
adequacy, 10
Agrawal, Mukesh
  on type annotations, 5
annotations
  contextual typing, 61
anti-colon, 177
asserting types, 30
  subtyping, 30

Baldwin, G.
  on having done it, 233
basis, 173
bidirectional typechecking, 233
bit vectors, 239
bitstrings, 198–204
block, 169
Booleans
  index sort, 219–220
brittle subterm, 137

Cayenne, 11
CFA, 166
colocated bindings, 137
compilation, 118
completeness
  of the left tridirectional system, 83
  of the tridirectional system, 65
completeness theorem
  left tridirectional system, 84
  let-normal system, 165
  tridirectional system, 71
constructor types, 24
context
  of constraint solver, 181
context-free languages
  as index domain, 241
contextual subtyping
  reflexivity, 65
contextual typing annotations, 61, 177
continuation-passing style, *see* CPS
control flow analysis, *see* CFA
convex theory, 190
Cooperating Validity Checker, *see* CVC
CPS, 118, 166
Curry-Howard isomorphism, 8, 56
CVC, 184
CVC Lite, 184

decidability of typing
  in the left tridirectional system, 87
definite types, 19
definiteness, 36
  theorem, 38
dependent types, 11–12
derivation measure μ, 37
derivation rank, 36–37
Dijkstra, Edsger W.
  on success, 233
dimensions

in object-oriented languages, 228
index sort, 220–230
direct style, 127
disjunctive propositions, 189–191
disordered linear variables, 158
dynamic typing, 8–9

empty type, 29
ephemeral refinements, 230
Epigram, 11
equivalence relations
index equality, 23
error reporting, 195–196
erstwhile CMU grad students, vii
erstwhile officemates, vii
ESC, 12
evaluation contexts $\mathcal{E}$, 19
exceptions, 170
existential dependent types, *see* existentially quantified types
existentially quantified types, 30
Extended Static Checker, *see* ESC
extends relation (terms), 67
reflexivity, 67
transitivity, 67

filter function, 27
focusing, 186
Forsythe, 191
functional arrays, 240

Girard, Jean-Yves
calling people morons, 9
on intersection types, 10
gratuitous citations, 187
greatest type, 21, 57
guarded types, 26
gutters, 233

ICS, 184
idempotency, 8
identity substitution, 35
impotence, 8
indefinite types, 27
index domain
properties
consequence, 23
equality $\doteq$, 23
substitution, 23
weakening, 23
induction measure
for let-normal completeness, 160
inductive families
as index domain, 239
injection phase (in Stardust), 179
integers
index sort, 197–219
intersection types, 21
invaluable refinements, 113, 220–230
inverse let-normal translation, *see* unwinding

left tridirectional system
completeness, 83
completeness theorem, 84
decidability of typing, 87
soundness of, 81
soundness theorem, 81
let-equivalence $\equiv_{\mathsf{let}}$, 127, 129
let-free viable paths $\mathcal{W}$, 135
let-free viable position, 136
let-normal completeness, 165
let-normal soundness, 134
let-normal translation
in Stardust, 181
inverse, *see* unwinding
traditional, 118
Xi's, 116
let-respecting, 132
letification, 138–143
light extends relation (terms), 67
light extension lemma, 68
linear logic, 186
linearity, 80
local type inference, 88, 236
ludics, 10

Mars Climate Orbiter, 226
maximal decomposition, 137
memoization, 187

minimal relevant logic, 49
monotonicity under annotation, 68
mutable references, 18, 242

natural numbers
  index sort, 198
negation of propositions, 175
Nelson-Oppen method, 242
non-convex theory, 190

object-oriented languages
  union types in, 49
object-oriented programming, 191
operational semantics, 20
  call by name, 242–244
  call by need, 244
  call by value, 20
operator overloading
  in Standard ML, 192
operator precedence
  in Stardust, 171
ordered sets
  as index domain, 240

parametric polymorphism, 196, 234–238
pattern matching
  in the let-normal system, 166
pattern typing
  compared to Davies' system, 112
  compared to Standard ML, 112
patterns
  implementation, 112, 189
permutation, 143–153
phantom types, 228
poetic language
  dreadful, 234
  questionable, 233
pointing hand notation "☞", 15
polymorphism, *see* intersection types, *see* parametric polymorphism
precedes relation (of subterms), 136
preservation, 44
prickly, 137
principal typings, 89

product sort flattening phase, 180
program extraction, 8
progress, 44

qualified types, 228

random colleagues, vii
rank, 36–37
red-black trees
  deletion, 209–219
  insertion, 204–209
redeemable bindings, 159
refinement restriction, 7, 191, 235
reflexivity
  of $\doteq$, 23
  of contextual subtyping, 65
  of subtyping, 34
  of the extends relation (terms), 67
regular languages
  as index domain, 241
relevant logic, 49
rewriting systems, 242
root (in a term), 137

semantics of refinements, 10
sets
  as index domain, 240
Shostak's method, 242
slack variables, 124, 188
slackening, 143
soft typing, 8–9
software engineering
  great moments in, 226
solver context, 181
soundness
  of pattern checking, 106
  of the left tridirectional system, 81
  of the tridirectional system, 65
soundness theorem
  left tridirectional system, 81
  let-normal system, 134
  tridirectional system, 65
Stanford Validity Checker, *see* SVC
steps-to relation $\mapsto$, 20

subset sorts, 27, 175
  asserting types, 30
  elimination phase (in Stardust), 180
  guarded types, 26
substitution
  identity, 35
  of let-bindings ($[L]e$), 130
  property of index domains, 23
  typing rules, 35
substitution lemma, 35
  ranked formulation, 38
subtyping
  decidability, 34
  reflexivity, 34
  summary of rules, 33
  transitivity, 34
SVC, 184
symmetry
  of $\doteq$, 23
synthesis subtyping, 24
synthesizing form, 67

tainted data, 228
termination checking, 10
top type, 21, 57
  subtyping, 57
transitivity
  of $\doteq$, 23
  of subtyping, 34
  of the extends relation (terms), 67
transposed linear variables, 138
tridirectional system
  completeness of, 65
  soundness of, 65
type annotations, 169, 177
type preservation, 44
type safety, 44
  left tridirectional system, 88
type safety proof, 45–48
type safety theorem, 45
  full pattern language, 110

unfortunate misfeatures
  of the implementation, 189
uninterpreted functions, 239
uninterpreted sets with order, *see* ordered sets
union types, 28
  for parametric polymorphism, 236
  subtyping, 28
universal dependent types, *see* universally quantified types
universally quantified types, 22
unwinding, 129

value definiteness, 36
  theorem, 38
value inversion, 41
value inversion on $\rightarrow$, $*$, $\delta(i)$, 41
value monotonicity, 34
value permutation, 153–157
values, 19
viable subterm, 121

weakening
  property of index domains, 23

Yices, 241