Tridirectional Typechecking

Jana Dunfield
jd169@queensu.ca

Frank Pfenning
fp@cs.cmu.edu

Carnegie Mellon University
Pittsburgh, PA

ABSTRACT

In prior work we introduced a pure type assignment system that encompasses a rich set of property types, including intersections, unions, and universally and existentially quantified dependent types. This system was shown sound with respect to a call-by-value operational semantics with effects, yet is inherently undecidable.

In this paper we provide a decidable formulation for this system based on bidirectional checking, combining type synthesis and analysis following logical principles. The presence of unions and existential quantification requires the additional ability to visit subterms in evaluation position before the context in which they occur, leading to a *tridirectional* type system. While soundness with respect to the type assignment system is immediate, completeness requires the novel concept of *contextual type annotations*, introducing a notion from the study of principal typings into the source program.

Categories and Subject Descriptors: F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Type structure*; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms: Languages, Theory

Keywords: Type refinements, intersection types, union types, dependent types

1. INTRODUCTION

Over the last two decades, there has been a steady increase in the use of type systems to capture program properties such as control flow [15], memory management [22], aliasing [20], data structure invariants [11, 7, 28] and effects [21, 14], to mention just a few. Ideally, such type systems specify rigorously, yet at a high level of abstraction, how to reason about a certain class of program properties. This specification usually serves a dual purpose: it is used to relate the properties of interest to the operational semantics of the programming language (for example, proving type preservation), and it is the basis for concrete algorithms for program analysis (for example, via constraint-based type inference).

While the type-based approach has been successful for use in automatic program analysis (for example, for optimization during compilation), it has been less successful in making the expressive type systems directly available to the programmer. One reason for this is the difficulty of finding the right balance between the brevity of the additional required type declarations and the feasibility of the typechecking problem. Another is the difficulty of giving precise and useful feedback to the programmer on ill-typed programs.

In prior work [9] we developed a system of pure type assignment designed for call-by-value languages with effects and proved progress and type preservation. The intended atomic program properties are data structure refinements [11, 10, 28], but our approach does not depend essentially on this choice. Atomic properties can be combined into more complex ones through intersections, unions, and universal and existential quantification over index domains. As a pure type assignment system, where terms do not contain any types at all, it is inherently undecidable [4].

In this paper we develop an annotation discipline and typechecking algorithm for our earlier 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 [13, 25]. 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* [18] of programs containing some type annotations, combining *type synthesis* with *type analysis*, first adapted to property types by Davies and Pfenning [7]. 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 what we call *indefinite property types*: unions and existential quantification. This is because the program lacks the prerequisite structure. For example, if we synthesize $A \vee B$, the union of A and 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. 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 di-

POPL'04, January 14–16, 2004, Venice, Italy.

rection" (whence the name *tridirectional*): we might need to move to a subexpression, synthesize its type, and only then analyze the expression 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 notion of type annotation $(e : A)$ (as previously noted [16, 6, 23]), 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 plan to explore in 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 brought out if the term containing it has the form $E[e]$ for some *evaluation context* E, reducing the nondeterminism; this was a key observation in our earlier paper [9]. Second, one never needs to visit a subterm more than once in the same derivation: the system which enforces this is sound and complete.

The remainder of the paper is organized as follows. Section 2 presents a simple bidirectional type system. Section 3 adds refinements and a rich set of types including intersections and unions, using tridirectional rules; this is the *simple tridirectional system*. In Section 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 5 restricts the tridirectional rules and compensates by introducing *left rules* to yield a *left tridirectional system*. We prove soundness and completeness with respect to the simple tridirectional system, prove decidability, and use the results in [9] to prove type safety. Finally, we discuss related work (Section 6) and conclude (Section 7).

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. The most straightforward 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 the bidirectional system we propose here. The motivation for this is twofold. First, for highly expressive systems such as the ones under consideration here, full type inference is often undecidable, so we 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. 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.

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

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.

$$\frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{fst}(e) \uparrow A}\ (*E_1) \qquad \frac{\Gamma \vdash e \uparrow A * B}{\Gamma \vdash \mathbf{snd}(e) \uparrow B}\ (*E_2)$$

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 $(\to\mathrm{I})$ checks against $A \to B$ and rule $(\to E)$ synthesizes the type $A \to B$ of its subject e_1.

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

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

$$\frac{\Gamma \vdash e \uparrow A' \quad \Gamma \vdash A' \leq A}{\Gamma \vdash e \downarrow A}\ (\mathsf{sub})$$

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). 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 4. For now, only normal terms will typecheck in our system. These correspond exactly to normal proofs in natural deduction. 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

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

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

$$
\begin{array}{ll}
(\lambda x.\, e)\, v \;\mapsto_R\; [v/x]\, e & \mathbf{fst}(v_1, v_2) \;\mapsto_R\; v_1 \\
\mathbf{fix}\, u.\, e \;\mapsto_R\; [\mathbf{fix}\, u.\, e \,/\, u]\, e & \mathbf{snd}(v_1, v_2) \;\mapsto_R\; v_2 \\
\multicolumn{2}{c}{\mathbf{case}\; c(v)\; \mathbf{of}\; \ldots c(x) \Rightarrow e \ldots \;\mapsto_R\; [v/x]\, e}
\end{array}
$$

Figure 1: Syntax and semantics of the core language

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.

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

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

We have elided here a syntactic condition that the left-hand sides of a **case** expression with subject δ cover all constructors for a type δ. Note that in the elimination rule (δE), we move from $e \uparrow \delta$ to $x{:}A$ (which may be read $x{\uparrow}A$), checking each branch against B.

In addition we have fixed points, which involve both directions: to check $\mathbf{fix}\, u.\, e \downarrow A$, we assume $u{:}A$ (which should be read $u{\uparrow}A$) and 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 $\mathbf{fix}\; u.\; e$ to $[\mathbf{fix}\; u.\; e \;/\; u]\, e$ (the substitution of $\mathbf{fix}\; u.\; e$ 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.

The syntax and semantics of our core language is given in Figure 1. A capital E denotes an evaluation context—a term with a hole $[\,]$ representing the part of the term where a reduction may occur. The semantics is a straightforward call-by-value small-step formulation. $[e'/x]\, e$ denotes the substitution of e' for x in e.

Figure 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 [26] used a contrasting style, in which several introduction forms have synthesis rules as well as checking rules, for example:

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

Xi's formulation reduces the number of annotations to some extent; for example, in **case** (x, y) **of** $\ldots$ 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 clear 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. 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 4; otherwise, only the language of types is enriched:

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

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 paper we do not formally distinguish between ordinary types and property types, though such a distinction has been useful in the study of refinement types [11, 10].

Our formulation of property types is fully explained and justified in [9] 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 follows directly from the result for the type assignment system [9].

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.

$$
\frac{\Gamma \vdash v \downarrow A \qquad \Gamma \vdash v \downarrow B}{\Gamma \vdash v \downarrow A \wedge B}\;(\wedge\mathrm{I})
$$

$$
\frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow A}\;(\wedge\mathrm{E}_1) \qquad \frac{\Gamma \vdash e \uparrow A \wedge B}{\Gamma \vdash e \uparrow B}\;(\wedge\mathrm{E}_2)
$$

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 [7].

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

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

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

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

$$\frac{c : A \to \delta \quad \Gamma \vdash e \downarrow A}{\Gamma \vdash c(e) \downarrow \delta}\ (\delta\mathrm{I}) \qquad \frac{\Gamma \vdash e \uparrow \delta \quad \Gamma \vdash ms \downarrow_\delta B}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ ms \downarrow B}\ (\delta\mathrm{E}) \qquad \frac{}{\Gamma \vdash \cdot \downarrow_\delta B} \qquad \frac{c : A \to \delta \quad \Gamma, x{:}A \vdash e \downarrow B \quad \Gamma \vdash ms \downarrow_\delta B}{\Gamma \vdash c(x) \Rightarrow e \mid ms \downarrow_\delta B}$$

Figure 2: Subtyping and typing in the core language

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. In fact, ignoring Γ for now, we can think of subtyping as a single-antecedent, single-succedent form of the sequent calculus.

$$\frac{\Gamma \vdash A \leq B_1 \quad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\ (\wedge\mathrm{R})$$

$$\frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\mathrm{L}_1) \quad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\ (\wedge\mathrm{L}_2)$$

We omit the common distributivity rule relating intersection and function types, which is unsound with mutable references [7] and does not directly fit into the logical pattern of our rules.

3.2 Greatest Type

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

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

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 [9].

3.3 Refined Datatypes

In our system, each datatype is refined as in [6, 8, 9] by an *atomic subtyping* relation $\preceq$ over *datasorts* δ. Each datasort identifies a subset of values of the form $c(v)$. For example, datasorts true and false identify singleton subsets of values of the type bool. We further refine datatypes by indices drawn from some constraint domain, exactly as in [9] which closely followed Xi and Pfenning [28], Xi [26, 27], and Dunfield [8]. The type $\delta(i)$ is the type of values having datasort δ and index i.

To accommodate index refinements, we extend Γ to allow *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 3). No variable may be declared twice in Γ, but ordering is now significant because of dependencies.

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 P must hold. Because

$$\begin{array}{rcl} P & ::= & \bot \mid i \doteq j \mid \ldots \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P \end{array} \qquad \begin{array}{rcl} \overline{\cdot} & = & \cdot \\ \overline{\Gamma, x{:}A} & = & \overline{\Gamma} \\ \overline{\Gamma, a{:}\gamma} & = & \overline{\Gamma}, a{:}\gamma \\ \overline{\Gamma, P} & = & \overline{\Gamma}, P \end{array}$$

Figure 3: Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

we have both universal and existential quantifiers over elements of the constraint domain, the constraints must remain decidable in the presence of quantifiers, though we have not encountered quantifier alternations in our examples. There must also be a relation $i \doteq j$ denoting index equality, and a judgment $\overline{\Gamma} \vdash i : \gamma$ whose interpretation is that i has *index sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have index sorts; terms and indices are distinct. The proof of safety in [9] requires that $\models$ be a consequence relation, that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that $\models$ and $\vdash$ have expected substitution and weakening properties [8].

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. In this paper, the only index sort is the natural numbers $\mathcal{N}$ with $\doteq$ and the arithmetic operations $+, -, *$. Then $\overline{\Gamma} \models P$ is decidable provided the equalities 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 function type on indices (instead of arbitrary terms).

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 if the list's length is odd, by a datasort even if it is even. We also refine the lists by their length, so Nil has type $\mathbf{1} \to \mathsf{even}(0)$, and Cons has type $(\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{even}(a) \to \mathsf{odd}(a+1)) \wedge (\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{odd}(a) \to \mathsf{even}(a+1))$. Writing $\mathsf{Nil}()$ as Nil, the function

$$\begin{array}{l} \mathbf{fix}\ \mathit{repeat}.\ \lambda x. \\ \quad \mathbf{case}\ x\ \mathbf{of}\ \mathsf{Nil} \Rightarrow \mathsf{Nil} \mid \mathsf{Cons}(h,t) \Rightarrow \mathsf{Cons}(h, \mathsf{Cons}(h, \mathit{repeat}(t))) \end{array}$$

checks against $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$.

The subtyping rule for datatypes checks the datasorts δ_1, δ_2 and (separately) the indices i, j:

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

To maintain reflexivity and transitivity of subtyping, we require $\preceq$ to be reflexive and transitive.

We assume the constructors c are typed by a judgment $\overline{\Gamma} \vdash c : A \to \delta(i)$ where A is any type and $\delta(i)$ is some refined type. Now, however, the type $A \to \delta(i)$ need not be unique; indeed, a constructor should often have more than one refined type. The rule for constructor application is

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

To derive $\Gamma \vdash$ **case** e **of** $ms \downarrow B$, we check that all the matches in ms check against B, under a context appropriate to each arm; this is how propositions P arise. The context Γ may be contradictory ($\overline{\Gamma} \models \bot$) if the case arm can be shown to be unreachable by virtue of the index refinements of the constructor type and the case subject. In order to not typecheck unreachable arms, we have

$$\frac{\overline{\Gamma} \models \bot}{\Gamma \vdash e \downarrow A} \ (\mathsf{contra})$$

We also do not check case arms that are unreachable by virtue of the *datasort* refinements. For a complete accounting of how we type **case** expressions and constructors, see [8].

The typing rules for Π are

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

By our general assumption, 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 for intersections, the introduction rule is restricted to values in order 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 4.

The subtyping rules for Π are

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

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 $A \leq B$ for an arbitrary $b{:}\gamma$ then A is also a subtype of $\Pi b{:}\gamma.\, B$. Of course, b cannot occur free in A.

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.

3.4 Indefinite Property Types

We now have a system with definite types $\wedge$, $\top$, Π. 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 standard example is the *filter* function on lists of integers: *filter* $f\ l$ returns the elements of l for which f returns true. It has the ordinary type *filter* : $(\mathsf{int}{\to}\mathsf{bool}) \to \mathsf{list} \to \mathsf{list}$. Indexing lists by their length, the refined type should look like

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

To fill in the blank, we add dependent sums $\Sigma a{:}\gamma.\, A$, quantifying existentially over index variables, as in [28, 26]. Then we can express the fact that *filter* returns a list of some indefinite length m as follows[1]:

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

For similar reasons, we also occasionally would like the union types and the empty type, which should also be considered indefinite. We discuss unions first.

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.

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

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 [9] without a syntactic marker[2] requires an evaluation context E around the subterm of union type.

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

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 E and subterm e'. Using the analysis and synthesis judgments we have

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

Here, if we can synthesize a union type for e'—which is in evaluation position in $E[e']$—and check $E[x]$ and $E[y]$ against C, assuming that x and y have type A and type B respectively, we can conclude that $E[e']$ 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 E and e', leading to excessive nondeterminism.

The subtyping rules are standard and dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \quad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B} \ (\vee\mathrm{L})$$

$$\frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2} \ (\vee\mathrm{R}_1) \qquad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2} \ (\vee\mathrm{R}_2)$$

The 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):

$$\frac{\Gamma \vdash e' \uparrow \bot}{\Gamma \vdash E[e'] \downarrow C} \ (\bot\mathrm{E})$$

As before, the expression must be an evaluation context E with e' in evaluation position. For $\top$ we had one right subtyping rule; for $\bot$, following the principle of duality, we have one left rule:

[1] The additional constraint $m \leq n$ could be expressed by a *subset sort* [27, 26].

[2] Pierce [17] 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.

$$\frac{}{\Gamma \vdash \bot \leq A}\ (\bot\text{L})$$

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

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

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

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

Again, there is a potentially subtle issue: the index variable a must be new and cannot be mentioned in an annotation in E.

The subtyping for Σ is dual to that of Π.

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

3.5 Properties of Subtyping

Our subtyping rules are the same as in [9] except for the addition of products $A * B$. Since the premises are smaller than the conclusion in each rule, and we assume decidability for the constraint domain, we immediately obtain that $\Gamma \vdash A \leq B$ is decidable. Reflexivity and transitivity are admissible, which follows quite easily [9].

3.6 The Tridirectional Rule

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

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

One might expect this rule to be admissible. However, due to the restriction to evaluation contexts, it is not. As a simple example, consider

$$\begin{array}{rl} \textit{append} & : \ \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(a+b) \\ \textit{filterpos} & : \ \Pi n{:}\mathcal{N}.\, \mathsf{list}(n) \to \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ & \vdash \ \textit{filterpos}\ [\ldots] \uparrow \Sigma m{:}\mathcal{N}.\, \mathsf{list}(m) \\ \text{Goal:} & \not\vdash \ \textit{append}\ [42]\ (\textit{filterpos}\ [\ldots]) \downarrow \Sigma k{:}\mathcal{N}.\, \mathsf{list}(k) \end{array}$$

where [42] is shorthand for $\mathsf{Cons}(42, \mathsf{Nil})$ and [...] is some literal list. Here we cannot derive the goal, because we cannot introduce the k on the type 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] will need to be evaluated first. However, *append* [42] synthesizes only type $\Pi b{:}\mathcal{N}.\, \mathsf{list}(b) \to \mathsf{list}(1{+}b)$, so we are stuck. However, using rule (direct) we reduce

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

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* [...]) is in evaluation position. Applying the existential elimination rule, we need to derive

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

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

4. CONTEXTUAL TYPING ANNOTATIONS

Our tridirectional system so far has the property that only terms in normal form have types. For example, $(\lambda x.\, x)\,()$ neither synthesizes nor checks against a type. This is because the function part of an application must synthesize a type, but there is no rule for $\lambda x.\, e$ to synthesize a type.

But annotations are not as straightforward as they might seem. In our setting, two issues arise: checking against intersections, and index variable scoping.

4.1 Checking Against Intersections

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

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

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.

Pierce [16] and Reynolds [19] addressed this problem by allowing a function to be annotated with a list of alternative types; the typechecker chooses the right one. Davies followed this approach in his datasort refinement checker, allowing a term to be annotated with $(e : A, B, \ldots)$. In that notation, the above function could be written as

$$\begin{array}{rl} \textit{cons42} \ = & (\lambda x.\, ((\lambda y.\, \mathsf{Cons}(42, x)) : \mathbf{1} \to \mathsf{even}, \mathbf{1} \to \mathsf{odd})\,()) \\ & : (\mathsf{odd} \to \mathsf{even}) \wedge (\mathsf{even} \to \mathsf{odd}) \end{array}$$

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 nondeterminism, since the typechecker must guess which type to use.

4.2 Index Variable Scoping

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

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

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

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

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

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

but not

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

Xi already noticed this problem and introduced a term-level abstraction over index variables, $\Lambda a.e$, to mirror universal index quantification $\Pi a{:}\gamma.\,A$ [26]. But this violates the basic principle of property types that the term should remain unchanged, and fails in the presence of intersections. For example, we would expect the reverse function on lists, *rev*, to satisfy

$$\begin{aligned} rev \;:\; & (\Pi a{:}\mathcal{N}.\,\mathsf{list}(a) \to \mathsf{list}(a)) \\ & \wedge ((\Sigma b{:}\mathcal{N}.\,\mathsf{list}(b)) \to \Sigma c{:}\mathcal{N}.\,\mathsf{list}(c)) \end{aligned}$$

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

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:

$$e \;::=\; \ldots \mid (e : \Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n)$$

where each context Γ_k declares the types of some, but not necessarily all, free variables in e.

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 $\mathsf{odd} \not\leq \mathsf{even}$).

$$\begin{aligned} x{:}\mathsf{even} \vdash ((\lambda y.\,\mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}) \uparrow \mathbf{1} \to \mathsf{odd} \\ x{:}\mathsf{odd} \not\vdash ((\lambda y.\,\mathsf{Cons}(42, x)) : x{:}\mathsf{even} \vdash \mathbf{1} \to \mathsf{odd}, \\ x{:}\mathsf{odd} \vdash \mathbf{1} \to \mathsf{even}) \uparrow \mathbf{1} \to \mathsf{odd} \end{aligned}$$

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

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

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

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

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 object 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, the free term variables in Γ_0 may actually occur in e and so cannot be renamed freely.

These considerations lead 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

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

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

Figure 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 5: Contextual subtyping

where we regard the annotations 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 them), in which case we need to try another annotation $(\Gamma_k \vdash A_k)$.

The formal rules for contextual subtyping are given in Figure 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 6 presented in the style of ML; we have omitted the evident constructor types and, following the tradition of implementations such as Davies', written typing annotations inside bracketed comments.

The essence of the completeness result we prove in Section 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, $(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$. Furthermore, in a judgment

$$\Gamma \vdash (e : (\Gamma_1 \vdash A_1, \ldots, \Gamma_n \vdash A_n)) \uparrow A$$

we must be able to consistently rename index variables in Γ, all Γ_k, and e. This different treatment of index variables and term variables arises from the fact that index variables are associated with property types and so do not appear in expressions, only in types.

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 1. $(\Gamma_2 \vdash A) \lesssim (\Gamma_1, \Gamma_2 \vdash A)$.

$$\mathsf{true} \preceq \mathsf{bool}, \quad \mathsf{false} \preceq \mathsf{bool} \qquad \mathsf{even} \preceq \mathsf{nat}, \quad \mathsf{odd} \preceq \mathsf{nat}$$

$$\mathsf{evenlist} \preceq \mathsf{list}, \quad \mathsf{oddlist} \preceq \mathsf{list}, \quad \mathsf{emptylist} \preceq \mathsf{evenlist}$$

$$\begin{aligned}
&\mathit{eq} : (\mathsf{even} * \mathsf{odd} \to \mathsf{false}) \\
&\quad \wedge (\mathsf{odd} * \mathsf{even} \to \mathsf{false}) \\
&\quad \wedge (\mathsf{nat} * \mathsf{nat} \to \mathsf{bool})
\end{aligned}$$

$$\begin{aligned}
&(*[\ \ \mathit{member} : (\mathsf{even} * \mathsf{oddlist} \to \mathsf{false}) \\
&\qquad\qquad \wedge (\mathsf{odd} * \mathsf{evenlist} \to \mathsf{false}) \\
&\qquad\qquad \wedge (\mathsf{nat} * \mathsf{list} \to \mathsf{bool}) \ \ ]*) \\
&\textbf{fun}\ \mathit{member}\ (x, xs) = \\
&\quad (*[\ \ \mathit{mem} : x{:}\mathsf{even} \vdash (\mathsf{evenlist} \to \mathsf{bool}) \wedge (\mathsf{oddlist} \to \mathsf{false}), \\
&\qquad\qquad x{:}\mathsf{odd} \vdash (\mathsf{evenlist} \to \mathsf{false}) \wedge (\mathsf{oddlist} \to \mathsf{bool}), \\
&\qquad\qquad x{:}\mathsf{nat} \vdash \mathsf{natlist} \to \mathsf{bool} \ \ ]*) \\
&\quad \textbf{let fun}\ \mathit{mem}\ xs = \\
&\qquad \textbf{case}\ xs\ \textbf{of}\ \mathsf{Nil} \Rightarrow \mathsf{False} \\
&\qquad \mid \mathsf{Cons}(y, ys) \Rightarrow \mathit{eq}(x, y)\ \textbf{orelse}\ \mathit{mem}\ ys \\
&\quad \textbf{in}\ \mathit{mem}\ xs\ \textbf{end}
\end{aligned}$$

$$\begin{aligned}
&(*[\ \ \mathit{append} : \Pi a{:}\mathcal{N}.\, \Pi b{:}\mathcal{N}.\, \mathsf{list}(a) * \mathsf{list}(b) \to \mathsf{list}(a+b) \ \ ]*) \\
&\textbf{fun}\ \mathit{append}\ (xs, ys) = \\
&\quad (*[\ \ \mathit{app} : c{:}\mathcal{N}, ys{:}\mathsf{list}(c) \vdash \Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{list}(a+c) \ \ ]*) \\
&\quad \textbf{let fun}\ \mathit{app}\ xs = \textbf{case}\ xs\ \textbf{of}\ \mathsf{Nil} \Rightarrow ys \\
&\qquad \mid \mathsf{Cons}(x, xs) \Rightarrow \mathsf{Cons}(x, \mathit{app}\ xs) \\
&\quad \textbf{in}\ \mathit{app}\ xs\ \textbf{end}
\end{aligned}$$

Figure 6: Example of contextual annotations

PROOF. By induction on Γ_2. □

COROLLARY 2 (REFLEXIVITY). $(\Gamma \vdash A) \lesssim (\Gamma \vdash A)$.

4.4 Soundness

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

THEOREM 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. □

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 12, a special case of Theorem 11) we need a few definitions and lemmas.

DEFINITION 4. *A term is in* synthesizing form *if it has any of the forms* x, $e_1 e_2$, u, $(e : As)$, $\mathbf{fst}(e)$, $\mathbf{snd}(e)$.

DEFINITION 5. e' extends *a term* e*, written* $e' \sqsupseteq e$ *iff* e' *is* e *with zero or more additional typing annotations and* e' *contains no type annotations on the roots of terms in synthesizing form.*

DEFINITION 6. 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*. That is, we can replace* $(e : As)$ *with* $(e : As, A')$*, but cannot replace* e *with* $(e : A')$.

PROPOSITION 7. $\sqsupseteq$ *and* $\sqsupseteq_\ell$ *are reflexive and transitive.*

PROOF. Obvious from the definitions. □

LEMMA 8. *If* e value *and* $e' \sqsupseteq e$ *then* e' value*.*

PROOF. By a straightforward induction on e' (in the base case, making use of $(v : As)$ value). □

LEMMA 9 (LIGHT EXTENSION). *If* $e' \sqsupseteq_\ell e$ *then (1)* $\Gamma \vdash e \uparrow A$ *implies* $\Gamma \vdash e' \uparrow A$*, (2)* $\Gamma \vdash e \downarrow A$ *implies* $\Gamma \vdash e' \downarrow A$.

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 $\mathbf{1}$. Note that this further annotation was light—we added a typing to an existing annotation. This observation leads to Lemma 10.

LEMMA 10 (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* $\Gamma \vdash e'' \uparrow A$.

PROOF. By induction on the typing derivation. □

THEOREM 11 (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* $\Gamma \vdash e''_2 \uparrow A$

PROOF. By induction on the derivation of $\Gamma \vdash e : A$. □

COROLLARY 12. *If* $\Gamma \vdash e : A$ *then there exists* $e' \sqsupseteq e$ *such that* $\Gamma \vdash e' \downarrow A$ *and there exists* $e'' \sqsupseteq e$ *such that* $\Gamma \vdash e'' \uparrow A$.

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

Subtyping	$\Gamma \vdash A \leq B$	Simple tridirectional checking	$\Gamma \vdash e \downarrow A$
Contextual subtyping	$(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A)$	Simple tridirectional synthesis	$\Gamma \vdash e \uparrow A$
Constraint satisfaction	$\overline{\Gamma} \models P$	Left tridirectional checking	$\Gamma; \Delta \vdash e \downarrow_{\mathbb{L}} A$
Index expression sorting	$\overline{\Gamma} \vdash i : \gamma$	Left tridirectional synthesis	$\Gamma; \Delta \vdash e \uparrow_{\mathbb{L}} A$
Data constructor typing	$\Gamma \vdash c : A \to \delta(i)$	Δ appear linearly in e	$\Delta \Vdash e$
		—and in evaluation position in e	$\Delta \mathrel{|\!|\!|\!-} e$

Figure 7: Judgment forms appearing in the paper

Rules of the simple tridirectional system absent in the left tridirectional system:

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

$$\frac{\Gamma \vdash e' \uparrow \bot}{\Gamma \vdash E[e'] \downarrow C}\ (\bot\mathrm{E})$$

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

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

Rules new or substantially altered in the left tridirectional system:

$$\frac{}{\Gamma; \overline{\mathbf{x}}{:}A \vdash \overline{\mathbf{x}} \uparrow_{\mathbb{L}} A}\ (\overline{\mathsf{var}})$$

$$\frac{e' \text{ not a linear var} \qquad \Gamma; \Delta_1 \vdash e' \uparrow_{\mathbb{L}} A \qquad \Gamma; \Delta_2, \overline{\mathbf{x}}{:}A \vdash E[\overline{\mathbf{x}}] \downarrow_{\mathbb{L}} C}{\Gamma; \Delta_1, \Delta_2 \vdash E[e'] \downarrow_{\mathbb{L}} C}\ (\mathsf{direct}\mathbb{L})$$

$$\frac{\Delta, \overline{\mathbf{x}}{:}\bot \Vdash e}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\bot \vdash e \downarrow_{\mathbb{L}} C}\ (\bot\mathbb{L})$$

$$\frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow_{\mathbb{L}} C \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow_{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vee B \vdash e \downarrow_{\mathbb{L}} C}\ (\vee\mathbb{L})$$

$$\frac{\Gamma, a{:}\gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow_{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\Sigma a{:}\gamma.\, A \vdash e \downarrow_{\mathbb{L}} C}\ (\Sigma\mathbb{L})$$

$$\frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \vdash e \downarrow_{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \downarrow_{\mathbb{L}} C}\ (\wedge\mathbb{L}_1) \qquad \frac{\Gamma; \Delta, \overline{\mathbf{x}}{:}B \vdash e \downarrow_{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}A \wedge B \vdash e \downarrow_{\mathbb{L}} C}\ (\wedge\mathbb{L}_2)$$

$$\frac{\overline{\Gamma} \vdash i : \gamma \qquad \Gamma; \Delta, \overline{\mathbf{x}}{:}[i/a]A \vdash e \downarrow_{\mathbb{L}} C}{\Gamma; \Delta, \overline{\mathbf{x}}{:}\Pi a{:}\gamma.\, A \vdash e \downarrow_{\mathbb{L}} C}\ (\Pi\mathbb{L})$$

Rules of the left tridirectional system identical to the simple tridirectional system, except for the linear contexts Δ:

$$\frac{\Gamma(x) = A}{\Gamma; \cdot \vdash x \uparrow A}\ (\mathsf{var}) \qquad \frac{\Gamma, x{:}A; \cdot \vdash e \downarrow B}{\Gamma; \cdot \vdash \lambda x.\, e \downarrow A \to B}\ (\to\mathrm{I}) \qquad \frac{\Gamma; \Delta_1 \vdash e_1 \uparrow A \to B \qquad \Gamma; \Delta_2 \vdash e_2 \downarrow A}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 e_2 \uparrow B}\ (\to\mathrm{E})$$

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

$$\frac{}{\Gamma; \cdot \vdash () \downarrow \mathbf{1}}\ (\mathbf{1}\mathrm{I}) \qquad \frac{\Gamma; \Delta_1 \vdash e_1 \downarrow A_1 \qquad \Gamma; \Delta_2 \vdash e_2 \downarrow A_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) \downarrow A_1 * A_2}\ (*\mathrm{I}) \qquad \frac{\Gamma; \Delta \vdash e \uparrow A * B}{\Gamma; \Delta \vdash \mathbf{fst}(e) \uparrow A}\ (*\mathrm{E}_1) \qquad \frac{\Gamma; \Delta \vdash e \uparrow A * B}{\Gamma; \Delta \vdash \mathbf{snd}(e) \uparrow B}\ (*\mathrm{E}_2)$$

$$\frac{\overline{\Gamma} \vdash c : A \to \delta_2(i) \qquad \Gamma \vdash \delta_2(i) \leq \delta_1(j) \qquad \Gamma; \Delta \vdash e \downarrow A}{\Gamma; \Delta \vdash c(e) \downarrow \delta_1(j)}\ (\delta\mathrm{I}) \qquad \frac{\overline{\Gamma} \models \bot \qquad \Delta \Vdash e}{\Gamma; \Delta \vdash e \downarrow A}\ (\mathsf{contra})$$

$$\frac{\Gamma; \Delta \vdash e \uparrow \delta(i) \qquad \Gamma; \cdot \vdash ms \downarrow_{\delta(i)} B}{\Gamma; \Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ ms \downarrow B}\ (\delta\mathrm{E})$$

$$\frac{\Delta \Vdash v}{\Gamma; \Delta \vdash v \downarrow \top}\ (\top\mathrm{I}) \qquad \frac{\Gamma; \Delta \vdash v \downarrow A \qquad \Gamma; \Delta \vdash v \downarrow B}{\Gamma; \Delta \vdash v \downarrow A \wedge B}\ (\wedge\mathrm{I}) \qquad \frac{\Gamma; \Delta \vdash e \uparrow A \wedge B}{\Gamma; \Delta \vdash e \uparrow A}\ (\wedge\mathrm{E}_1) \qquad \frac{\Gamma; \Delta \vdash e \uparrow A \wedge B}{\Gamma; \Delta \vdash e \uparrow B}\ (\wedge\mathrm{E}_2)$$

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

$$\frac{\Gamma; \Delta \vdash e \downarrow A}{\Gamma; \Delta \vdash e \downarrow A \vee B}\ (\vee\mathrm{I}_1) \qquad \frac{\Gamma; \Delta \vdash e \downarrow B}{\Gamma; \Delta \vdash e \downarrow A \vee B}\ (\vee\mathrm{I}_2) \qquad \frac{(\Gamma_0 \vdash A_0) \lesssim (\Gamma \vdash A) \qquad \Gamma \vdash e \downarrow A}{\Gamma \vdash (e : (\Gamma_0 \vdash A_0), As) \uparrow A}\ (\mathsf{ctx\text{-}anno})$$

Figure 8: The left tridirectional system, with the part of the simple tridirectional system (upper left corner) from which it substantially differs. 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 $\Delta \Vdash e$, from the lower rules, along with the rules in the upper left corner. Hence the subscripts $\uparrow_{\mathbb{L}}$, $\downarrow_{\mathbb{L}}$ are elided.

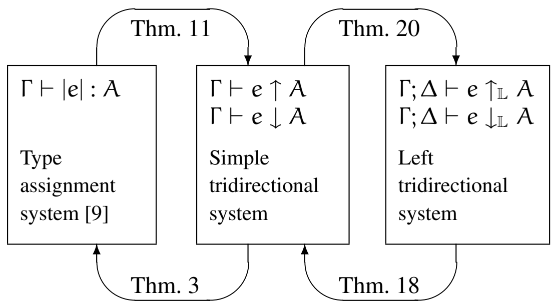

Figure 9: Connections between our type systems

section has only one contextual rule and disallows repeated application. Inspired by the sequent calculus formulation of Barbanera et al. [2], it replaces the contextual rules with one contextual rule ($\mathsf{direct}\mathbb{L}$), closely corresponding to (direct), and several *left rules*, shown in the upper right hand corner of Figure 8. In combination, these rules subsume the contextual rules of the simple tridirectional system.

The typing judgments in the left tridirectional system are

$$\Gamma; \Delta \vdash e \uparrow_{\mathbb{L}} A \qquad \Gamma; \Delta \vdash e \downarrow_{\mathbb{L}} A$$

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.

The rule ($\mathsf{direct}\mathbb{L}$) is the only rule that adds to the linear context, and is the true source of linearity: $\bar{\mathbf{x}}$ appears exactly once in evaluation position in $E[\bar{\mathbf{x}}]$. 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 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, in rules such as ($\rightarrow$I) that type terms that cannot contain a variable, the linear context is empty.

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. (See also Figure 9).

DEFINITION 13. *Let* $FLV(e)$ *denote the set of linear variables appearing free in* e. *Furthermore, let* $\Delta \Vdash e$ *if and only if (1) for every* $\bar{\mathbf{x}} \in \mathrm{dom}(\Delta)$, $\bar{\mathbf{x}}$ *appears exactly once in* e, *and (2)* $FLV(e) \subseteq \mathrm{dom}(\Delta)$. *(Similarly define* $FLV(ms)$ *and* $\Delta \Vdash ms$.*)*

PROPOSITION 14 (LINEARITY). *If* $\Gamma; \Delta \vdash e \uparrow_{\mathbb{L}} C$ *or* $\Gamma; \Delta \vdash e \downarrow_{\mathbb{L}} C$ *then* $\Delta \Vdash e$. *Similarly, if* $\Gamma; \Delta \vdash ms \downarrow_{\delta(i)} C$ *then* $\Delta \Vdash ms$.

PROOF. By induction on the derivation. For (contra), ($\top$I), ($\bot\mathbb{L}$), use the appropriate premise. □

DEFINITION 15. *Let* $\Delta \Vvdash e$ *if and only if (1) for every* $\bar{\mathbf{x}} \in \mathrm{dom}(\Delta)$, *there exists an* E *such that* $e = E[\bar{\mathbf{x}}]$ *and* $\bar{\mathbf{x}} \notin FLV(E)$, *and (2)* $FLV(e) \subseteq \mathrm{dom}(\Delta)$. *(It is clear that* $\Delta \Vvdash e$ *implies* $\Delta \Vdash e$.*)*

LEMMA 16. *If* $\mathcal{D}$ *derives* $\Gamma; \Delta \vdash e \uparrow_{\mathbb{L}} C$ *or* $\Gamma; \Delta \vdash e \downarrow_{\mathbb{L}} C$ *by a rule* R *and* $\Delta \Vvdash e$, *then for each premise* $\Gamma'; \Delta' \vdash e' \uparrow_{\mathbb{L}} C'$ *or* $\Gamma'; \Delta' \vdash e' \downarrow_{\mathbb{L}} C'$ *of* R, *it is the case that* $\Delta' \Vvdash e'$.

PROOF. Straightforward. □

5.1 Soundness

DEFINITION 17. *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, $\rho = x/\bar{\mathbf{x}}, \ldots$, 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 $\mathrm{dom}(\rho)$.

THEOREM 18 (SOUNDNESS, LEFT RULE 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* $\Delta \Vvdash e$ *and* $\mathrm{dom}(\rho) \supseteq \mathrm{dom}(\Delta)$, *then* $\Gamma, [\rho]\Delta \vdash [\rho]e \uparrow C$ *(resp.* $\Gamma, [\rho]\Delta \vdash [\rho]e \downarrow C$*).*

The condition $\Delta \Vvdash e$ 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 use Lemma 16 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.

For ($\overline{\mathsf{var}}$), it is given that $\mathrm{dom}(\rho) \supseteq \mathrm{dom}(\Delta)$, so we can apply ($\mathsf{var}$). For ($\mathsf{direct}\mathbb{L}$), use the IH on the first premise, let x be new, and use the IH on the second premise with the renaming $\rho, x/\bar{\mathbf{x}}$; apply properties of substitution and weakening to yield derivations to which (direct) can be applied. For the left rules, use a different renaming $\rho, x'/\bar{\mathbf{x}}$ where x' is new for each premise, then apply the IH to yield derivation(s) typing $[\rho, x'/\bar{\mathbf{x}}]\, E[\bar{\mathbf{x}}]$ (by $\Delta \Vvdash e$, $e = E[\bar{\mathbf{x}}]$). Use (var) to obtain a typing of $[\rho]\bar{\mathbf{x}}$. Finally, apply the corresponding tridirectional rule, such as ($\vee$E) for the ($\vee\mathbb{L}$) case. □

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, a small lemma:

LEMMA 19. *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. □

THEOREM 20 (COMPLETENESS, LEFT RULE SYSTEM). *If* ρ *renames ordinary program variables to linear variables and* $\Gamma \vdash e \uparrow C$ *(resp.* $\Gamma \vdash e \downarrow C$*) and* $\Delta \Vvdash [\rho]e$ *where* $[\rho]\Gamma = \Gamma'; \Delta$, *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 of the cases can be handled as follows: Restrict ρ to variables appearing in subterms of e (if any). Apply the IH to all premises. Reason that if ρ' is a restriction of ρ to a subterm e', then the result of applying the IH—namely $[\rho']\Gamma \vdash [\rho']e' \downarrow_{\mathbb{L}} A$—implies $[\rho]\Gamma \vdash [\rho]e' \downarrow_{\mathbb{L}} A$. Finally, reapply the original rule.

However, this fails for the rules that are absent or modified in the left tridirectional system: (direct), ($\bot$E), ($\vee$E), (ΣE). In each of the cases for these rules, there are two subcases:

- If the subterm e' is not a variable renamed by ρ, then we apply the IH to the premise typing e', make a new linear variable $\bar{\mathbf{x}}$, apply the IH to the contextual premises as needed, apply the corresponding left rule (or do nothing in the (direct) case) to show $E[\bar{\mathbf{x}}] \downarrow_{\mathbb{L}} C$, then apply ($\mathsf{direct}\mathbb{L}$).

- If e' is a variable in $\mathrm{dom}(\rho)$, we apply the IH to all premises, apply the corresponding left rule (or do nothing in the (direct) case), then use Lemma 19. □

5.3 Decidability of Typing

THEOREM 21. *$\Gamma; \Delta \vdash e \downarrow_{\mathbb{L}} A$ is decidable.*

PROOF. We impose an order $<$ on two judgments $\mathcal{J}_1 = \Gamma_1; \Delta_1 \vdash e_1 \downarrow\uparrow A_1$ and $\mathcal{J}_2 = \Gamma_2; \Delta_2 \vdash e_2 \downarrow\uparrow A_2$. When ordering terms, we consider linear variables to be smaller than any other terms; for example, $(\bar{\mathbf{x}}, e_2)$ is smaller than (y, e_2)). When ordering types (that is, type expressions), we consider all index expressions to be of equal size.

The order is defined as follows.

1. If e_1 is smaller than e_2 then $\mathcal{J}_1 < \mathcal{J}_2$. If e_1 is the same size as e_2:

2. If the directions of the judgments differ, the synthesis judgment is smaller than the checking judgment. If the directions are the same:

3. 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 as small as, or smaller than, some type in $(\Gamma_1; \Delta_1)$, and A_1 is *larger* than A_2, then $\mathcal{J}_1 < \mathcal{J}_2$. Otherwise:

4. If the number of times any of the type constructors $\vee$, Σ, $\bot$, $\wedge$, Π, $\top$ 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. For most premises, the first criterion alone makes the premise smaller. The second criterion is for (sub). The third 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 ($*E_1$), 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 fourth 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. □

5.4 Type Safety

If $\cdot; \cdot \vdash e \downarrow_{\mathbb{L}} A$ in the left tridirectional system, from Theorem 18 we know $\cdot \vdash e \downarrow A$. Then by Theorem 3, $\cdot \vdash |e| : A$ in our type assignment system [9]. That is, type erasure suffices to get a typing derivation in the type assignment system. It follows from [9]'s Theorem 3, Type Preservation and Progress, that $|e|$ either diverges or evaluates to a value of type A.

6. RELATED WORK

Refinements, intersections, unions. The notion of datasort refinement combined with intersection types was introduced by Freeman and Pfenning [11]. They showed that full type inference was decidable under the so-called refinement restriction by using techniques from abstract interpretation. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning [7] who introduced the value restriction on intersection introduction, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm.

Index refinements were proposed by Xi and Pfenning [28]. As mentioned earlier, the necessary existential quantifier Σ led to difficulties [26] 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. Nonetheless, we conjecture that Xi's idea of traversing the entire program strictly in evaluation order is applicable in our significantly more complex setting to eliminate the nondeterminism inherent in the (direct$\mathbb{L}$) rule; we plan to pursue this in further research.

Intersection types [4] were first incorporated into a practical language by Reynolds [19]. Pierce [17] gave examples of programming with intersection and union types in a pure λ-calculus using a typechecking mechanism that relied on syntactic markers. The first systematic study of unions in a type assignment framework [2] identified several issues, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted. It also provided a framework for our more specialized study of a call-by-value language with possible effects.

Some work on program analysis in compilation uses intersection and union types to infer control flow properties [24, 15]. 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 ours. *Soft typing* systems designed for type inference under dynamic typing [3] are somewhat similar, allowing intersection, union, and even conditional types [1]. Again, due to the different setting and goal, the technical realization differs substantially from our work.

Partial inference systems. Our system shares several properties with Pierce and Turner's *local type inference* [18]. 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. However, in order to handle parametric polymorphism without using nonlocal methods such as unification, they infer type arguments to polymorphic functions, which seems to substantially complicate matters. Hosoya and Pierce [12] further discuss this style, particularly its effectiveness in achieving a reasonable number of annotations.

Our system does not yet have parametric polymorphism. Prior research, either with (in [26]) or without (in [7]) a syntactic distinction between ordinary and property types, is not conclusive. However, the work on local type inference suggests that, at least, prefix polymorphism in the style of ML should be amenable to a consistent treatment with bidirectional rules.

Principal typings. A *principal type* of e is a type that represents all types of e—in some particular context Γ. A *principal typing* [13] 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 [25] improved the situation by introducing a general notion of representation. Since full type inference seems in any case unattainable, we have not investigated whether principal typings might 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.

7. CONCLUSION

In [9], we 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. In this paper, 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 should be useful in other settings, such as systems of parametric polymorphism in which subtyping is decidable.

In order to show decidability, and as a 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.

We are in the process of formulating a let-normal version of the left tridirectional system. Such a system would drastically reduce the nondeterminism in (direct$\mathbb{L}$) by forcing the typechecker to traverse subterms in evaluation order, while being sound and complete with respect to the left tridirectional system.

Once this is done, we plan to develop a prototype implementation of the let-normal system that should help us answer questions regarding the practicality of our design on realistic programs. The main questions will be (1) if the required annotations are reasonable in size, (2) if type checking is efficient enough for interesting program properties, and (3) if the typing discipline is accurate enough to track properties in complex programs. The preliminary experience with refinement types, including both datasort refinements [5] and index refinements [28], gives reason for optimism, but more research and experimentation is needed.

Acknowledgments. This work supported in part by the National Science Foundation under grant CCR-0204248; the first author was also supported in part by an NSF Graduate Research Fellowship. We thank Jonathan Moody, Sungwoo Park, and the anonymous reviewers for their useful comments. We also thank Rowan Davies for many fruitful discussions regarding the subject of this paper.

8. REFERENCES

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *ACM Symp. Principles of Programming Languages*, pages 163–173, 1994.

[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.

[3] R. Cartwright and M. Fagan. Soft typing. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991.

[4] 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.

[5] 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.

[6] Rowan Davies. Practical refinement-type checking. PhD thesis proposal, Carnegie Mellon University, 1997.

[7] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000.

[8] Jana Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002.

[9] Jana Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computational Structures (FOSSACS '03)*, pages 250–266, Warsaw, Poland, April 2003. Springer LNCS 2620.

[10] Tim Freeman. *Refinement types for ML*. PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-110.

[11] Tim Freeman and Frank Pfenning. Refinement types for ML. In *SIGPLAN Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 268–277. ACM Press, 1991.

[12] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[13] Trevor Jim. What are principal typings and what are they good for? Technical memorandum MIT/LCS/TM-532, MIT, November 1995.

[14] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. Technical Report TR-656-02, Princeton, December 2002.

[15] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Func. Prog.*, 11(3):263–317, 2001.

[16] Benjamin C. Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-205.

[17] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[18] 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.

[19] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[20] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *European Symp. on Programming (ESOP'00)*, pages 366–381, Berlin, Germany, March 2000.

[21] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. and Comp.*, 111(2):245–296, 1994.

[22] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. and Comp.*, 132(2):109–176, 1997.

[23] J. B. Wells and Christian Haack. Branching types. In *European Symp. on Programming (ESOP'02)*, pages 115–132, 2002.

[24] J.B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Func. Prog.*, 12(3):183–317, May 2002.

[25] Joe Wells. The essence of principal typings. In *Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

[26] Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.

[27] Hongwei Xi. Dependently typed data structures. Revision superseding WAAAPL '99, February 2000.

[28] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symp. Principles of Programming Languages*, pages 214–227, 1999.