Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types

JANA DUNFIELD, Queen's University, Canada
NEELAKANTAN R. KRISHNASWAMI, University of Cambridge, United Kingdom

Bidirectional typechecking, in which terms either synthesize a type or are checked against a known type, has become popular for its applicability to a variety of type systems, its error reporting, and its ease of implementation. Following principles from proof theory, bidirectional typing can be applied to many type constructs. The principles underlying a bidirectional approach to indexed types (*generalized algebraic datatypes*) are less clear. Building on proof-theoretic treatments of equality, we give a declarative specification of typing based on *focalization.* This approach permits declarative rules for coverage of pattern matching, as well as support for first-class existential types using a focalized subtyping judgment. We use refinement types to avoid explicitly passing equality proofs in our term syntax, making our calculus similar to languages such as Haskell and OCaml. We also extend the declarative specification with an explicit rules for deducing when a type is principal, permitting us to give a complete declarative specification for a rich type system with significant type inference. We also give a set of algorithmic typing rules, and prove that it is sound and complete with respect to the declarative system. The proof requires a number of technical innovations, including proving soundness and completeness in a mutually recursive fashion.

Additional Key Words and Phrases: bidirectional typechecking, higher-rank polymorphism, indexed types, GADTs, equality types, existential types

1 INTRODUCTION

Consider a list type Vec with a numeric index representing its length, in Agda-like notation:

$$\begin{array}{l} \texttt{data}\ \mathsf{Vec} : \mathsf{Nat} \to \mathsf{Type} \to \mathsf{Type}\ \texttt{where} \\ \quad \texttt{[]} \ :\ A \to \mathsf{Vec}\ 0\ A \\ \quad (::)\ :\ A \to \mathsf{Vec}\ n\ A \to \mathsf{Vec}\ (\mathsf{succ}\ n)\ A \end{array}$$

We can use this definition to write a head function that always gives us an element of type A when the length is at least one:

$$\begin{array}{l} \texttt{head} : \forall n, A.\ \mathsf{Vec}\ (\mathsf{succ}\ n)\ A \to A \\ \texttt{head}\ (x :: xs) = x \end{array}$$

This clausal definition omits the clause for `[]`, which has an index of 0. The type annotation tells us that `head`'s argument has an index of $\mathsf{succ}(n)$ for some n. Since there is no natural number n such that $0 = \mathsf{succ}(n)$, the nil case cannot occur and can be omitted.

This is an entirely reasonable explanation for programmers, but language designers and implementors will have more questions. First, designers of functional languages are accustomed to the

Authors' addresses: Jana Dunfield, Queen's University, Goodwin Hall 557, Kingston, ON, K7L 3N6, Canada, jd169@queensu.ca; Neelakantan R. Krishnaswami, University of Cambridge, Computer Laboratory, William Gates Building, Cambridge, CB3 0FD, United Kingdom, nk480@cl.cam.ac.uk.

2475-1421/2019/1-ART9
https://doi.org/10.1145/3290322

benefits of the Curry–Howard correspondence, and expect a *logical* reading of type systems to accompany the operational reading. So what is the logical reading of GADTs? Second, how can we implement such a type system? Clearly we needed some equality reasoning to justify leaving off the nil case, which is not trivial in general.

Since we relied on equality information to omit the nil case, it seems reasonable to look to logical accounts of equality. In proof theory, it is possible to formulate equality in (at least) two different ways. The better-known is the *identity type* of Martin-Löf, but GADTs actually correspond best to the equality of Schroeder-Heister [1994] and Girard [1992]. The Girard–Schroeder-Heister (GSH) approach introduces equality via the reflexivity principle:

$$\frac{}{\Gamma \vdash t = t}$$

The GSH elimination rule was originally formulated in a sequent calculus style, as follows:

$$\frac{\text{for all } \theta.\text{ if } \theta \in \mathsf{csu}(s,t) \text{ then } \theta(\Gamma) \vdash \theta(C)}{\Gamma, (s = t) \vdash C}$$

Here, we write $\mathsf{csu}(s,t)$ for a *complete set of unifiers* of s and t. So the rule says that we can eliminate an equality $s = t$ if, for every substitution θ that makes s and t equal, we can prove the goal C.

This rule has three important features, two good and one bad.

(1) The GSH elimination rule is an invertible left rule. By "left rule", we mean that the rule decomposes the assumptions to the left of the turnstile (in this case, the assumption that $s = t$), and by "invertible", we mean the conclusion of the rule implies the premises.[1]
Invertible left rules are interesting, because they are known to correspond (via Curry–Howard) to the deconstruction steps carried out by pattern-matching rules [Krishnaswami 2009]. This is our first hint that the GSH rule has something to do with GADTs; programming languages like Haskell and OCaml indeed use pattern matching to propagate equality information.

(2) Observe that if we have an inconsistent equation, we can immediately prove the goal. If we specialize the rule above to the equality $0 = 1$, we get:

$$\frac{}{\Gamma, (0 = 1) \vdash C}$$

Because 0 and 1 have no unifiers, the complete set of unifiers is the empty set. As a result, the GSH rule for this instance has no premises, and the elimination rule for an absurd equation ends up looking exactly like the elimination rule for the empty type:

$$\frac{}{\Gamma, \bot \vdash C}$$

Moreover, recall that when we eliminate an empty type, we can view the eliminator $\mathsf{abort}(e)$ as a pattern match with no clauses. Together, these two features line up nicely with our definition of `head`, where the impossibility of the case for `[]` was indicated by the *absence* of a pattern clause. The use of equality in GADTs corresponds perfectly with the GSH equality.

(3) Alas, we cannot simply give a proof term assignment for first-order logic and call it a day. The third important feature of the GSH equality rule is its use of *unification*: it works by treating the free variables of the two terms as unification variables. But type inference algorithms also use unification, introducing unification variables to stand for unknown types.
These two uses of unification are *entirely different!* Type inference introduces unification variables to stand for the specific instantiations of universal quantifiers. In contrast, the Girard–Schroeder-Heister rule uses unification to constrain the universal parameters. As a

[1] The invertibility of equality elimination is certainly not obvious; see Schroeder-Heister [1994] and Girard [1992].

result, we need to understand how to integrate these two uses of unification, or at least how to keep them decently separated, in order to take this logical specification and implement type inference for it.

This problem—formulating indexed types in as logical a style as feasible, while retaining the ability to implement type inference algorithms for them—is the subject of this paper.

Contributions. It has long been known that GADTs are equivalent to the combination of existential types and equality constraints [Xi et al. 2003]. Our key contribution is to reduce GADTs to standard logical ingredients, *while retaining the implementability of the type system*. We manage this by formulating a system of indexed types in a bidirectional style (combining type *synthesis* with *checking* against a known type), which is both practically implementable and theoretically tidy.

- Our language supports implicit higher-rank polymorphism (in which quantifiers can be nested under arrows) including existential types. While algorithms for higher-rank universal polymorphism are well-known [Peyton Jones et al. 2007; Dunfield and Krishnaswami 2013], our approach to supporting existential types is novel.
  We go beyond the standard practice of tying existentials to datatype declarations [Läufer and Odersky 1994], in favour of a first-class treatment of implicit existential types. This approach has historically been thought difficult, since treating existentials in a first-class way opens the door to higher-rank polymorphism that mixes universal and existential quantifiers.
  Our approach extends existing bidirectional methods for handling higher-rank polymorphism, by adapting the proof-theoretic technique of *focusing* to give a novel *polarized subtyping judgment*, which lets us treat mixed quantifiers in a way that retains decidability while maintaining the essential properties of subtyping, such as stability under substitution and transitivity.
- Our language includes equality types in the style of Girard and Schroeder-Heister, but without an explicit introduction form for equality. Instead, we treat equalities as property types, in the style of intersection or refinement types: we do not write explicit equality proofs in our syntax, permitting us to more closely model how equalities are used in OCaml and Haskell.
- The use of focusing also lets us equip our calculus with nested pattern matching. This fits in neatly with our bidirectional framework, and permits us to give a formal specification of coverage checking with GADTs, which is easy to understand, easy to implement, and theoretically well-motivated.
- In contrast to systems which globally possess or lack principal types, our declarative system tracks whether or not a derivation has a principal type.
  Our system includes an unusual "higher-order principality" rule, which says that if only a single type can be synthesized for a term, then that type is principal. While this style of hypothetical reasoning is natural to explain to programmers, formalizing it requires giving an inference rule with universal quantification over possible typing derivations in the premise. This is an extremely non-algorithmic rule (even its well-foundedness is not immediate).
  As a result, the soundness and completeness proofs for our implementation have to be done in a new style. It is no longer possible to prove soundness and completeness independently, and instead we must prove them mutually recursively.
- We formulate an algorithmic type system (Section 5) for our declarative calculus, and prove that typechecking is decidable, deterministic (5.4), and sound and complete (Sections 6–7) with respect to the declarative system.
  The resulting type system is relatively easy to implement (an undergraduate implemented most of it on his own from a draft of the paper, with minimal contact with the authors), and

is close in style to languages such as Haskell or OCaml. As a result, it seems like a reasonable basis for implementing new languages with expressive type systems.

Our algorithmic system (and, to a lesser extent, our declarative system) uses some techniques developed by Dunfield and Krishnaswami [2013], but we extend these to a far richer type language (existentials, indexed types, sums, products, equations over type variables), and we differ by supporting pattern matching, polarized subtyping, and principality tracking.

Supplementary material. The appendix contains rules omitted for space reasons, and full proofs.

2 OVERVIEW

To orient the reader, we give an overview and rationale of the novelties in our type system, before getting into the details of the typing rules and algorithm. As is well-known [Cheney and Hinze 2003; Xi et al. 2003], GADTs can be desugared into type expressions that use equality and existential types to express the return type constraints. These two features lead to difficulties in typechecking for GADTs.

Universal, existentials, and type inference. Practical typed functional languages must support some degree of type inference, most critically the inference of type arguments. That is, if we have a function f of type $\forall a.\, a \to a$, and we want to apply it to the argument 3, then we want to write $f\ 3$, and not $f\ [\mathsf{Nat}]\ 3$ (as we would in pure System F). Even with a single type argument, the latter style is noisy, and programs using even moderate amounts of polymorphism rapidly become unreadable. However, omitting type arguments has significant metatheoretical implications. In particular, it forces us to include subtyping in our typing rules, so that (for instance) the polymorphic type $\forall a.\, a \to a$ is a subtype of its instantiations (like $\mathsf{Nat} \to \mathsf{Nat}$).

The subtype relation induced by System F-style polymorphism and function contravariance is already undecidable [Tiuryn and Urzyczyn 1996; Chrząszcz 1998], so even at the first step we must introduce restrictions on type inference to ensure decidability. In our case, matters are further complicated by the fact that we need to support *existential types* in addition to universal types.

Existentials are required to encode GADTs [Xi and Pfenning 1999], but programming languages have traditionally stringently restricted the use of existential types. Following the approach of Läufer and Odersky [1994], languages such as OCaml and Haskell tie existential introduction and elimination to datatype declarations, so that there is always a syntactic marker for when to introduce or eliminate existential types. This choice permits leaving existentials out of subtyping altogether, at the price of no longer permitting implicit subtyping (such as using $\lambda x.\, x + 1$ at type $\exists a.\, a \to a$).

While this is a practical solution, it increases the distance between a surface language and its type-theoretic core. Our goal is to give a *direct* type-theoretic account of as many features of our surface languages as possible. In addition to the theoretical tidiness, this also has practical language design benefits. By avoiding a complex elaboration step, we are forced to specify the type inference algorithm in terms of a language close to a programmer-visible surface language. This does increase the complexity of the approach, but in a productive way: we are forced to analyze and understand how type inference will look to the end programmer.

The key problem is that when both universal and existential quantifiers are permitted, the order in which to instantiate quantifiers when computing subtype entailments becomes unclear. For example, suppose we need to decide $\Gamma \vdash \forall a_1.\, \exists a_2.\, A(a_1, a_2) \le \exists b_1.\, \forall b_2.\, B(b_1, b_2)$. An algorithm to solve this must either first introduce a unification variable for a_1 and a parameter for a_2 first, and only then introduce a unification variable for b_1 and a parameter for b_2, or the other way around—and the order in which we make these choices matters! With the first order, the instantiation for

b_1 may refer to a_2, but the instantiation for a_1 cannot have b_2 as a free variable. With the second order, the instantiation for a_1 may have b_2 as a free variable, but b_1 may not refer to a_2.

In some cases, depending on what $A(a_1, a_2)$ and $B(b_1, b_2)$ are, only one choice of order works. For example, if we are trying to decide $\Gamma \vdash \forall a_1.\ \exists a_2.\ a_1 \to a_2 \leq \exists b_1.\ \forall b_2.\ b_2 \to b_1$, we must choose the first order: we must pick an instantiation for a_1, and then make a_2 into a parameter before we can instantiate b_1 as a_2. The second order will not work, because b_1 must depend on a_2. Conversely, if we are trying to solve $\Gamma \vdash \forall a_1.\ \exists a_2.\ a_1 \to a_2 \leq \exists b_1.\ \forall b_2.\ \exists b_3.\ b_1 \times b_2 \to b_3$, the first order will not work; we must instantiate b_1 (say, to int) and quantify over b_2 before instantiating a_1 as $\mathsf{int} \times b_2$.

As a result, the outermost connectives do not reliably determine which side of a subtype judgement $\Gamma \vdash \forall a.\ A \leq \exists b.\ B$ to specialize first.

One implementation strategy is simply to give up determinism: an algorithm could backtrack when faced with deciding subtype entailments of this form. Unfortunately, backtracking is dangerous for a practical implementation, since it potentially causes type-checking to take exponential time. This tends to defeat the benefit of a complete declarative specification, since different implementations with different backtracking strategies could have radically different running times when checking the same program. So we may end up with an implementation that is theoretically complete, but incomplete in practice.

Instead, we turn to ideas from proof theory—specifically, polarized type theory. In the language of polarization, universals are a *negative* type, and existentials are a *positive* type. So we introduce two mutually recursive subtype relations: $\Gamma \vdash A \leq^+ B$ for positive types and $\Gamma \vdash A \leq^- B$ for negative types. The positive subtype relation only deconstructs existentials, and the negative subtype relation only deconstructs universals. This fixes the order in which quantifiers are instantiated, making the problem decidable (in fact, rather straightforward).

The price we pay is that fewer subtype entailments are derivable. Fortunately, any program typeable under a more liberal subtyping judgement can be made typable in our discipline by η-expanding it. Moreover, the lost subtype entailments seem to be rare in practice: most of the types we see in practice are of the form $\forall \vec{a}.\ \vec{A} \to \exists \vec{b}.\ B$, and this fits perfectly with the kinds of types our polarized subtyping judgement works best on. As a result, we keep fundamental expressivity, and also efficient decidability.

Equality as a property. The usual convention in Haskell and OCaml is to make equality proofs in GADT definitions implicit. We would like to model this feature directly, so that our calculus stays close to surface languages, without sacrificing the logical reading of the system. In this case, the appropriate logical concepts come from the theory of intersection types. A typing judgment such as $e : A \times B$ can be viewed as giving instructions on how to construct a value (pair an A with a B). But types can also be viewed as *properties*, where $e : X$ is read "e has property X".

To model GADTs, we need both of these readings! For example, a term of vector type is constructed from nil and cons constructors, but also has a property governing its index. To support this combination, we introduce a type constructor $A \land P$, read "A with P", to model elements of type A satisfying the property (equation) P. (We also introduce $P \supset A$, read "P implies A", for its adjoint dual, consisting of terms which have the type A conditionally under the assumption that P holds.) Then we make equality $t = t'$ into a property, and make use of standard rules for property types (which omit explicit proof terms) to type equality constraints [Dunfield 2007b, Section 2.4].

This gives us a logical account of how OCaml and Haskell avoid requiring explicit equality proofs in their syntax. The benefit of handling equality constraints through intersection types is that certain restrictions on typing that are useful for decidability, such as restricting property introduction to values, arise naturally from the semantic point of view—via the value restriction needed for soundly modeling intersection and union types [Davies and Pfenning 2000; Dunfield and Pfenning 2003]. In addition, the appropriate approach to take when combining GADTs and effects is clear.[2]

Bidirectionality, pattern matching, and principality. Something that is not by itself novel in our approach is our decision to formulate both the declarative and algorithmic systems in a bidirectional style. Bidirectional checking [Pierce and Turner 2000] is a popular implementation choice for systems ranging from dependent types [Coquand 1996; Abel et al. 2008] and contextual types [Pientka 2008] to object-oriented languages [Odersky et al. 2001], but also has good proof-theoretic foundations [Watkins et al. 2004], making it useful both for specifying and implementing type systems. Bidirectional approaches make it clear to programmers where annotations are needed (which is good for specification), and can also remove unneeded nondeterminism from typing (which is good for both implementation and proving its correctness).

However, it is worth highlighting that because both bidirectionality and pattern matching arise from focalization, these two features fit together extremely well. In fact, by following the blueprint of focalization-based pattern matching, we can give a coverage-checking algorithm that explains when it is permissible to omit clauses in GADT pattern matching.

In the propositional case, the type synthesis judgment of a bidirectional type system generates principal types: if a type can be inferred for a term, that type is the most specific possible type for that term. (Indeed, in many cases, including the current system, the inferred type will even be unique.) This property is lost once quantifiers are introduced into the system, which is why it is not much remarked upon. However, prior work on GADTs, starting with Simonet and Pottier [2007], has emphasized the importance of the fact that handling equality constraints is much easier when the type of a scrutinee is principal. Essentially, this ensures that no existential variables can appear in equations, which prevents equation solving from interfering with unification-based type inference. The OutsideIn algorithm takes this consequence as a definition, permitting non-principal types just so long as they do not change the values of equations. However, Vytiniotis et al. [2011] note that while their system is sound, they no longer have a completeness result for their type system.

We use this insight to extend our bidirectional typechecking algorithm to track principality: The judgments we give track whether types are principal, and we use this to give a relatively simple specification for whether or not type annotations are needed. We are able to give a very natural spec to programmers—cases on GADTs must scrutinize terms with principal types, and an inferred type is principal just when it is the only type that can be inferred for that term—which soundly and completely corresponds to the implementation-side constraints: a type is principal when it contains no existential unification variables.

3 EXAMPLES

In this section, we give some examples of terms from our language, which illustrate the key features of our system and give a sense of how many type annotations are needed in practice. To help make this point clearly, all of the examples which follow are unsugared: they are the *actual* terms from our core calculus.

Mapping over lists. First, we begin with the traditional *map* function, which takes a function and applies it to every element of a list.

[2]The traditional `eq` GADT and its constructor `refl` can be encoded into our system as the type $1 \wedge (s = t)$, which which can be constructed as a unit value only under the constraint that s equals t.

$$\begin{array}{l}\texttt{rec}\ \mathit{map}.\,\lambda f.\,\lambda xs.\,\mathsf{case}\Big(xs, \quad \texttt{[]} \Rightarrow \texttt{[]} \\ \qquad\qquad\qquad\qquad\qquad\quad \mid y :: ys \Rightarrow (f\ y) :: \mathit{map}\ f\ ys\Big) \\ : \forall n : \mathbb{N}.\, \forall \alpha : \star.\, \forall \beta : \star.\, (\alpha \to \beta) \to \mathsf{Vec}\ n\ \alpha \to \mathsf{Vec}\ n\ \beta\end{array}$$

This function case-analyzes its second argument xs. Given an empty xs, it returns the empty list; given a cons cell $y :: ys$, it applies the argument function f to the head y and makes a recursive call on the tail ys.

In addition, we annotate the definition with a type. We have two type parameters α and β for the input and output element types. Since we are working with length-indexed lists, we also have a length index parameter n, which lets us show by typing that the input and output of *map* have the same length.

In our system, this type annotation is mandatory. Full type inference for definitions using GADTs requires polymorphic recursion, which is undecidable. As a result, this example also requires annotation in OCaml and GHC Haskell. However, Haskell and OCaml infer polymorphic types when no polymorphic recursion is needed. We adopt the simpler rule that *all* polymorphic definitions are annotated. This choice is motivated by Vytiniotis et al. [2010], who analyzed a large corpus of Haskell code and showed that implicit let-generalization was used primarily only for top-level definitions, and even then it is typically considered good practice to annotate top-level definitions for documentation purposes. Furthermore, experience with languages such as Agda and Idris (which do not implicitly generalize) show this is a modest burden in practice.

Nested patterns and GADTs. Now, we consider the *zip* function, which converts a pair of lists into a list of pairs. In ordinary ML or Haskell, we must consider what to do when the two lists are not the same length. However, with length-indexed lists, we can statically reject passing two lists of differing length:

$$\begin{array}{l}\texttt{rec}\ \mathit{zip}.\,\lambda p.\,\mathsf{case}\Big(p, \quad (\texttt{[]}, \texttt{[]}) \Rightarrow \texttt{[]} \\ \qquad\qquad\qquad\qquad\quad \mid (x :: xs, y :: ys) \Rightarrow (x, y) :: \mathit{zip}\ (xs, ys)\Big) \\ : \forall n : \mathbb{N}.\, \forall \alpha : \star.\, \forall \beta : \star.\, (\mathsf{Vec}\ n\ \alpha \times \mathsf{Vec}\ n\ \beta) \to \mathsf{Vec}\ n\ (\alpha \times \beta)\end{array}$$

This case expression has only two patterns, one for when both lists are empty and one for when both lists have elements, with the type annotation indicating that both lists must be of length n. Typing shows that the cases where one list is empty and the other is non-empty are impossible, so our coverage checking rules accept this as a complete set of patterns. This example also illustrates that we support nested pattern matching.

Existential Types. Now, we consider the *filter* function, which takes a predicate and a list, and returns a list containing the elements satisfying that predicate. This example makes a nice showcase for supporting existential types, since the size of the return value is not predictable statically.

$$\begin{array}{l}\texttt{rec}\ \mathit{filter}.\,\lambda p.\,\lambda xs.\ \mathsf{case}\Big(xs, \quad \texttt{[]} \Rightarrow \texttt{[]} \\ \qquad\qquad\qquad\qquad\qquad\quad \mid x :: xs \Rightarrow \mathsf{let}\ tl = \mathit{filter}\ p\ xs\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{case}\Big(p\ x, \quad \mathsf{inj}_1\ \_ \Rightarrow tl \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{inj}_2\ \_ \Rightarrow x :: tl\Big)\Big) \\ : \forall n : \mathbb{N}.\, \forall \alpha : \star.\, (\alpha \to 1 + 1) \to \mathsf{Vec}\ n\ \alpha \to \exists k : \mathbb{N}.\, \mathsf{Vec}\ k\ \alpha\end{array}$$

So, this function takes predicate and a vector of arbitrary size, and then returns a list of unknown size (represented by the existential type $\exists k : \mathbb{N}.\, \mathsf{Vec}\ k\ \alpha$). Note that we did not need to package the existential in another datatype, as one would have to in OCaml or GHC Haskell—we are free to use existential types as "just another type constructor".

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & x \mid () \mid \lambda x.\, e \mid e\; s^{+} \mid \mathsf{rec}\; x.\, v \mid (e : A) \\
& & & \mid \langle e_1, e_2 \rangle \mid \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid \mathsf{case}(e, \Pi) \\
& & & \mid [] \mid e_1 :: e_2 \\
\text{Values} & v & ::= & x \mid () \mid \lambda x.\, e \mid \mathsf{rec}\; x.\, v \mid (v : A) \\
& & & \mid \langle v_1, v_2 \rangle \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid [] \mid v_1 :: v_2 \\
\text{Spines} & s & ::= & \cdot \mid e\; s \\
\text{Nonempty spines} & s^{+} & ::= & e\; s \\
\text{Patterns} & \rho & ::= & x \mid \langle \rho_1, \rho_2 \rangle \mid \mathsf{inj}_1\, \rho \mid \mathsf{inj}_2\, \rho \mid [] \mid \rho_1 :: \rho_2 \\
\text{Branches} & \pi & ::= & \vec{\rho} \Rightarrow e \\
\text{Branch lists} & \Pi & ::= & \cdot \mid \left(\pi \mid \Pi\right)
\end{array}
$$

Fig. 1. Source syntax

$$
\begin{array}{lrcl}
\text{Universal variables} & \alpha, \beta, \gamma & & \\
\text{Sorts} & \kappa & ::= & \star \mid \mathbb{N} \\
\text{Types} & A, B, C & ::= & 1 \mid A \to B \mid A + B \mid A \times B \\
& & & \mid \alpha \mid \forall \alpha : \kappa.\, A \mid \exists \alpha : \kappa.\, A \\
& & & \mid P \supset A \mid A \wedge P \mid \mathsf{Vec}\; t\; A \\
\text{Terms/monotypes} & t, \tau, \sigma & ::= & \mathsf{zero} \mid \mathsf{succ}(t) \mid 1 \mid \alpha \\
& & & \mid \tau \to \sigma \mid \tau + \sigma \mid \tau \times \sigma \\
\text{Propositions} & P, Q & ::= & t = t' \\
\text{Contexts} & \Psi & ::= & \cdot \mid \Psi, \alpha : \kappa \mid \Psi, x : A\, p \\
\text{Polarities} & \mathcal{P} & ::= & + \mid - \mid \circ \\
\text{Binary connectives} & \oplus & ::= & \to \mid + \mid \times \\
\text{Principalities} & p, q & ::= & ! \mid \underbrace{\not{!}}_{\text{sometimes omitted}}
\end{array}
$$

Fig. 2. Syntax of declarative types and contexts

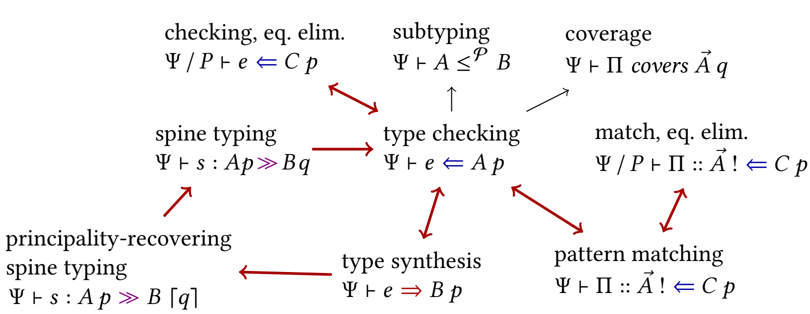

Fig. 3. Dependency structure of the declarative judgments

4 DECLARATIVE TYPING

Expressions. Expressions (Figure 1) are variables x; the unit value $()$; functions $\lambda x.\, e$; applications to a spine $e\; s^{+}$; fixed points $\mathsf{rec}\; x.\, v$; annotations $(e : A)$; pairs $\langle e_1, e_2 \rangle$; injections into a sum type

$\boxed{\begin{array}{l}\mathrm{pol}(A)\\ \mathit{nonpos}(A)\\ \mathit{nonneg}(A)\end{array}}$ Determine the polarity of a type / Check if A not positive / Check if A not negative

$$\begin{array}{lcl}
\mathrm{pol}(\forall\alpha:\kappa.\,A) & = & - \\
\mathrm{pol}(\exists\alpha:\kappa.\,A) & = & + \\
\mathrm{pol}(A) & = & \circ \text{ otherwise} \\
\mathit{nonpos}(A) & \text{iff} & \mathrm{pol}(A) \neq + \\
\mathit{nonneg}(A) & \text{iff} & \mathrm{pol}(A) \neq -
\end{array}$$

$\boxed{\mathrm{join}(\mathcal{P}_1, \mathcal{P}_2)}$ Join polarities

$$\begin{array}{lcl}
\mathrm{join}(+, \mathcal{P}_2) & = & + \\
\mathrm{join}(-, \mathcal{P}_2) & = & - \\
\mathrm{join}(\circ, +) & = & + \\
\mathrm{join}(\circ, -) & = & - \\
\mathrm{join}(\circ, \circ) & = & -
\end{array}$$

$\boxed{\Psi \vdash A \leq^{\mathcal{P}} B}$ Under context Ψ, type A is a subtype of B, decomposing head connectives of polarity $\mathcal{P}$

$$\frac{\Psi \vdash A\ \mathit{type} \qquad \mathit{nonpos}(A) \qquad \mathit{nonneg}(A)}{\Psi \vdash A \leq^{\mathcal{P}} A}\ \leq\mathsf{Refl}\mathcal{P}
\qquad
\frac{\Psi \vdash A \leq^{-} B \qquad \mathit{nonpos}(A) \qquad \mathit{nonpos}(B)}{\Psi \vdash A \leq^{+} B}\ \leq^{-}_{+}$$

$$\frac{\Psi \vdash A \leq^{+} B \qquad \mathit{nonneg}(A) \qquad \mathit{nonneg}(B)}{\Psi \vdash A \leq^{-} B}\ \leq^{+}_{-}$$

$$\frac{\Psi \vdash \tau : \kappa \qquad \Psi \vdash [\tau/\alpha]A \leq^{-} B}{\Psi \vdash \forall\alpha{:}\kappa.\, A \leq^{-} B}\ \leq\forall\mathsf{L}
\qquad
\frac{\Psi, \beta : \kappa \vdash A \leq^{-} B}{\Psi \vdash A \leq^{-} \forall\beta{:}\kappa.\, B}\ \leq\forall\mathsf{R}$$

$$\frac{\Psi, \alpha : \kappa \vdash A \leq^{+} B}{\Psi \vdash \exists\alpha{:}\kappa.\, A \leq^{+} B}\ \leq\exists\mathsf{L}
\qquad
\frac{\Psi \vdash \tau : \kappa \qquad \Psi \vdash A \leq^{+} [\tau/\beta]B}{\Psi \vdash A \leq^{+} \exists\beta{:}\kappa.\, B}\ \leq\exists\mathsf{R}$$

Fig. 4. Subtyping in the declarative system

$\mathsf{inj}_k\ e$; case expressions $\mathrm{case}(e, \Pi)$ where Π is a list of branches π, which can eliminate pairs and injections (see below); the empty vector $[]$; and consing a head e_1 to a tail vector e_2.

Values v are standard for a call-by-value semantics; the variables introduced by fixed points are considered values, because we only allow fixed points of values. A spine s is a list of expressions—arguments to a function. Allowing empty spines (written $\cdot$) is convenient in the typing rules, but would be strange in the source syntax, so (in the grammar of expressions e) we require a nonempty spine s^+. We usually omit the empty spine $\cdot$, writing $e_1\ e_2$ instead of $e_1\ e_2\ \cdot$. Since we use juxtaposition for both application $e\ s^+$ and spines, some strings are ambiguous; we resolve this ambiguity in favour of the spine, so $e_1\ e_2\ e_3$ is parsed as the application of e_1 to the spine $e_2\ e_3$, which is technically $e_2\ (e_3\ \cdot)$. Patterns ρ consist of pattern variables, pairs, and injections. A branch π is a *sequence* of patterns $\vec{\rho}$ with a branch body e. We represent patterns as sequences, which enables us to deconstruct tuple patterns.

Types. We write types as A, B and C. We have the unit type 1, functions $A \to B$, sums $A + B$, and products $A \times B$. We have universal and existential types $\forall\alpha : \kappa.\ A$ and $\exists\alpha : \kappa.\ A$; these are predicative quantifiers over monotypes (see below). We write α, β, etc. for type variables; these are universal, except when bound within an existential type. We also have a *guarded type* $P \supset A$, read "P implies A". This implication corresponds to type A, provided P holds. Its dual is the *asserting type* $A \land P$, read "A with P", which witnesses the proposition P. In both, P has no runtime content.

Sorts, terms, monotypes, and propositions. Terms and monotypes t, τ, σ share a grammar but are distinguished by their *sorts* κ. Natural numbers zero and $\mathrm{succ}(t)$ are *terms* and have sort $\mathbb{N}$. Unit 1 has the sort $\star$ of *monotypes*. A variable α stands for a term or a monotype, depending on the sort κ

annotating its binder. Functions, sums, and products of monotypes are monotypes and have sort $\star$. We tend to prefer t for terms and σ, τ for monotypes.

A proposition P or Q is simply an equation $t = t'$. Note that terms, which represent runtime-irrelevant information, are distinct from expressions; however, an expression may include type annotations of the form $P \supset A$ and $A \wedge P$, where P contains terms.

Contexts. A declarative context Ψ is an *ordered* sequence of universal variable declarations $\alpha : \kappa$ and expression variable typings $x : A\, p$, where p denotes whether the type A is principal (Section 4.2). A variable α can be free in a type A only if α was declared to the left: $\alpha : \star,\ x : \alpha\, p$ is well-formed, but $x : \alpha\, p,\ \alpha : \star$ is not.

4.1 Subtyping

We give our two subtyping relations, $\leq^+$ and $\leq^-$, in Figure 4. We treat the universal quantifier as a negative type (since it is a function in System F), and the existential as a positive type (since it is a pair in System F). We have two typing rules for each of these connectives, corresponding to the left and right rules for universals and existentials in the sequent calculus. We treat all other types as having no polarity. The positive and negative subtype judgments are mutually recursive, and the $\leq^-_+$ rule permits switching the polarity of subtyping from positive to negative when both of the types are non-positive, and conversely for $\leq^+_-$. When both types are neither positive nor negative, we require them to be equal ($\leq$Refl$\mathcal{P}$).

In logical terms, functions and guarded types are negative; sums, products and assertion types are positive. We could potentially operate on these types in the negative and positive subtype relations, respectively. Leaving out (for example) function subtyping means that we will have to do some η-expansions to get programs to typecheck; we omit these rules to keep the implementation complexity low. (The idea that η-expansion can substitute for subsumption dates to Barendregt et al. [1983].)

This also illustrates a nice feature of bidirectional typing: we are relatively free to adjust the subtype relation to taste. Moreover, the structure of polarization makes it easy to work out just what the rules should be. E.g., to add function subtyping to our system, we would use the rule:

$$\frac{\Psi \vdash A' \leq^+ A \qquad \Psi \vdash B \leq^- B'}{\Psi \vdash A \to B \leq^- A' \to B'}$$

As polarized function types are a negative type of the form $X^+ \to Y^-$, we see (1) the rule as a whole lives in the negative subtyping judgement, (2) argument types compare in the positive judgement (with the usual contravariant twist), and (3) result types compare in the negative judgement.

$\boxed{e\ \mathit{chk}\text{-}I}$ Expression e is a checked introduction form

$$\frac{}{\lambda x.\, e\ \mathit{chk}\text{-}I} \qquad \frac{}{()\ \mathit{chk}\text{-}I} \qquad \frac{}{\langle e_1, e_2\rangle\ \mathit{chk}\text{-}I} \qquad \frac{}{\mathsf{inj}_k\, e\ \mathit{chk}\text{-}I} \qquad \frac{}{[]\ \mathit{chk}\text{-}I} \qquad \frac{}{e_1 :: e_2\ \mathit{chk}\text{-}I}$$

Fig. 5. "Checking intro form"

4.2 Typing Judgments

Principality. Our typing judgments carry *principalities*: $A\,!$ means that A is principal, and $A\,\not{!}$ means A is not principal. Note that a principality is part of a judgment, not part of a type. In the checking judgment $\Psi \vdash e \Leftarrow A\ p$ the type A is input; if $p = !$, we know that A is not the result of guessing. For example, the e in $(e : A)$ is checked against $A\,!$. In the synthesis judgment

$\boxed{\Psi \vdash P\ \mathit{true}}$ Under context Ψ, check P

$$\frac{}{\Psi \vdash (t = t)\ \mathit{true}}\ \text{DeclCheckpropEq}$$

$\boxed{\begin{array}{l}\Psi \vdash e \Leftarrow A\,p \\ \Psi \vdash e \Rightarrow A\,p\end{array}}$ Under context Ψ, expression e checks against input type A; Under context Ψ, expression e synthesizes output type A

$$\frac{x : A\,p \in \Psi}{\Psi \vdash x \Rightarrow A\,p}\ \text{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A\,q \qquad \Psi \vdash A \leq^{\mathrm{join}(\mathrm{pol}(B),\mathrm{pol}(A))} B}{\Psi \vdash e \Leftarrow B\,p}\ \text{DeclSub}$$

$$\frac{\Psi \vdash A\ \mathit{type} \qquad \Psi \vdash e \Leftarrow A\,!}{\Psi \vdash (e : A) \Rightarrow A\,!}\ \text{DeclAnno} \qquad \frac{\Psi, x : A\,p \vdash v \Leftarrow A\,p}{\Psi \vdash \mathsf{rec}\ x.\,v \Leftarrow A\,p}\ \text{DeclRec} \qquad \frac{}{\Psi \vdash () \Leftarrow 1\,p}\ \text{Decl1I}$$

$$\frac{v\ \mathit{chk\text{-}I} \qquad \Psi, \alpha : \kappa \vdash v \Leftarrow A\,p}{\Psi \vdash v \Leftarrow (\forall \alpha : \kappa.\, A)\ p}\ \text{Decl}\forall\text{I} \qquad \frac{\Psi \vdash \tau : \kappa \qquad \Psi \vdash e \Leftarrow [\tau/\alpha]A}{\Psi \vdash e \Leftarrow (\exists \alpha : \kappa.\, A)\ p}\ \text{Decl}\exists\text{I}$$

$$\frac{v\ \mathit{chk\text{-}I} \qquad \Psi \,/\, P \vdash v \Leftarrow A\,!}{\Psi \vdash v \Leftarrow (P \supset A)\ !}\ \text{Decl}\supset\text{I} \qquad \frac{\Psi \vdash P\ \mathit{true} \qquad \Psi \vdash e \Leftarrow A\,p}{\Psi \vdash e \Leftarrow (A \wedge P)\ p}\ \text{Decl}\wedge\text{I}$$

$$\frac{\Psi, x : A\,p \vdash e \Leftarrow B\,p}{\Psi \vdash \lambda x.\, e \Leftarrow A \to B\ p}\ \text{Decl}\to\text{I} \qquad \frac{\Psi \vdash e \Rightarrow A\,p \qquad \Psi \vdash s : A\,p \gg C\ \lceil q \rceil}{\Psi \vdash e\ s \Rightarrow C\,q}\ \text{Decl}\to\text{E}$$

$$\frac{\Psi \vdash e \Rightarrow A\,q \qquad \Psi \vdash \Pi :: A\,! \Leftarrow C\,p \qquad \forall B.\ \text{if}\ \Psi \vdash e \Rightarrow B\,q\ \text{then}\ \Psi \vdash \Pi\ \mathit{covers}\ B\,q}{\Psi \vdash \mathsf{case}(e, \Pi) \Leftarrow C\,p}\ \text{DeclCase}$$

$\boxed{\begin{array}{l}\Psi \vdash s : A\,p \gg C\,q \\ \Psi \vdash s : A\,p \gg C\ \lceil q \rceil\end{array}}$ Under context Ψ, passing spine s to a function of type A synthesizes type C; in the $\lceil q \rceil$ form, recover principality in q if possible

$$\frac{\Psi \vdash \tau : \kappa \qquad \Psi \vdash e\ s : [\tau/\alpha]A\ \not{!} \gg C\,q}{\Psi \vdash e\ s : (\forall \alpha : \kappa.\, A)\ p \gg C\,q}\ \text{Decl}\forall\text{Spine} \qquad \frac{\Psi \vdash P\ \mathit{true} \qquad \Psi \vdash e\ s : A\,p \gg C\,q}{\Psi \vdash e\ s : (P \supset A)\ p \gg C\,q}\ \text{Decl}\supset\text{Spine}$$

$$\frac{}{\Psi \vdash \cdot : A\,p \gg A\,p}\ \text{DeclEmptySpine} \qquad \frac{\Psi \vdash e \Leftarrow A\,p \qquad \Psi \vdash s : B\,p \gg C\,q}{\Psi \vdash e\ s : A \to B\ p \gg C\,q}\ \text{Decl}\to\text{Spine}$$

$$\frac{\Psi \vdash s : A\,! \gg C\ \not{!} \qquad \begin{array}{l}\text{for all}\ C'.\\ \quad \text{if}\ \ \Psi \vdash s : A\,! \gg C'\ \not{!}\\ \quad \text{then}\ C' = C\end{array}}{\Psi \vdash s : A\,! \gg C\ \lceil ! \rceil}\ \text{DeclSpineRecover} \qquad \frac{\Psi \vdash s : A\,p \gg C\,q}{\Psi \vdash s : A\,p \gg C\ \lceil q \rceil}\ \text{DeclSpinePass}$$

$\boxed{\Psi \,/\, P \vdash e \Leftarrow C\,p}$ Under context Ψ, incorporate proposition P and check e against C

$$\frac{\mathrm{mgu}(\sigma, \tau) = \bot}{\Psi \,/\, (\sigma = \tau) \vdash e \Leftarrow C\,p}\ \text{DeclCheck}\bot \qquad \frac{\begin{array}{c}\mathrm{mgu}(\sigma, \tau) = \theta\\ \theta(\Psi) \vdash \theta(e) \Leftarrow \theta(C)\ p\end{array}}{\Psi \,/\, (\sigma = \tau) \vdash e \Leftarrow C\,p}\ \text{DeclCheckUnify}$$

Fig. 6. Declarative typing, omitting rules for $\times$, $+$, and Vec

$\Psi \vdash e \Rightarrow A\,p$, the type A is output, and $p = !$ means it is impossible to synthesize any other type, as in $\Psi \vdash (e : A) \Rightarrow A\,!$.

We sometimes omit a principality when it is $\not{!}$ (“not principal”). We write $p \sqsubseteq q$, read “p at least as principal as q”, for the reflexive closure of $! \sqsubseteq \not{!}$.

Spine judgments. The ordinary form of spine judgment, $\Psi \vdash s : A\, p \gg C\, q$, says that if arguments s are passed to a function of type A, the function returns type C. For a function e applied to one argument e_1, we write $e\ e_1$ as syntactic sugar for $e\ (e_1\ \cdot)$. Supposing e synthesizes $A_1 \to A_2$, we apply Decl$\to$Spine, checking e_1 against A_1 and using DeclEmptySpine to derive $\Psi \vdash \cdot : A_2\, p \gg A_2\, p$.

Rule Decl$\forall$Spine does not decompose $e\ s$ but instantiates a $\forall$. Note that, even if the given type $\forall \alpha : \kappa.\ A$ is principal ($p = \mathbf{!}$), the type $[\tau/\alpha]A$ in the premise is not principal—we could choose a different τ. In fact, the q in Decl$\forall$Spine is also always $\not{!}$, because no rule deriving the ordinary spine judgment can recover principality.

The *recovery spine judgment* $\Psi \vdash s : A\, p \gg C\, \lceil q \rceil$, however, can restore principality in situations where the choice of τ in Decl$\forall$Spine cannot affect the result type C. If A is principal ($p = \mathbf{!}$) but the ordinary spine judgment produces a non-principal C, we can try to recover principality with DeclSpineRecover. Its first premise is $\Psi \vdash s : A\, \mathbf{!} \gg C\, \not{!}$; its second premise (really, an infinite set of premises) quantifies over all derivations of $\Psi \vdash s : A\, \mathbf{!} \gg C'\, \not{!}$. If $C' = C$ in all such derivations, then the ordinary spine rules erred on the side of caution: C is actually principal, so we can set $q = \mathbf{!}$ in the conclusion of DeclSpineRecover.

If some $C' \neq C$, then C is certainly not principal, and we must apply DeclSpinePass, which simply transitions from the ordinary judgment to the recovery judgment.

Figure 3 shows the dependencies between the declarative judgments. Given the cycle containing the spine typing judgments, we need to stop and ask: Is DeclSpineRecover well-founded? For well-foundedness of type systems, we can often make a straightforward argument that, as we move from the conclusion of a rule to its premises, either the expression gets smaller, or the expression stays the same but the type gets smaller. In DeclSpineRecover, neither the expression nor the type get smaller. Fortunately, the rule that gives rise to the arrow from “spine typing” to “type checking” in Figure 3—Decl$\to$Spine—*does* decompose its subject, and any derivations of a recovery judgment lurking within the second premise of DeclSpineRecover must be for a smaller spine. In the appendix (Lemma 1, p. 38), we prove that the recovery judgment, and all the other declarative judgments, are well-founded.

Example. In Section 5.1 we present some example derivations that illustrate how the spine typing rules work to recover principality.

Subtyping. Rule DeclSub invokes the subtyping judgment, at the join of the polarities of B (the type being checked against) and A (the type being synthesized). Using the join ensures that the polarity of B takes precedence over A’s, which means the programmer control which subtyping mode to begin with via a type annotation.

Furthermore, the subtyping rule allows DeclSub to play the role of an existential introduction rule, by applying subtyping rule $\leq\exists$R when B is an existential type.

Pattern matching. Rule DeclCase checks that the scrutinee has a type and principality, and then invokes the two main judgments for pattern matching. The $\Psi \vdash \Pi :: \vec{A}\, q \Leftarrow C\, p$ judgement checks that each branch in the list of branches Π is well-typed, taking a vector $\vec{A}$ of pattern types to simplify the specification of coverage checking, as a well as a principality annotation covering all of the types (i.e., if any of the types in $\vec{A}$ is non-principal, the whole vector is not principal).

The $\Psi \vdash \Pi$ *covers* $\vec{A}\, q$ judgement does coverage checking for the list of branches. However, the DeclCase does not simply check that the patterns cover for the inferred type of the scrutinee—it checks that they cover for *every* possible type that could be inferred for the scrutinee. In the case that the scrutinee is principal, this is the same as checking coverage at the scrutinee’s type, but when the scrutinee is not principal, this rule has the effect of preventing type inference from using the shape of the patterns to infer a type, which is notoriously problematic with GADTs (e.g.,

$\boxed{\Psi \vdash \Pi :: \vec{A}\; q \Leftarrow C\; p}$ Under context Ψ, check branches Π with patterns of type $\vec{A}$ and bodies of type C

$$\frac{}{\Psi \vdash \cdot :: \vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatchEmpty} \qquad \frac{\Psi \vdash \pi :: \vec{A}\; q \Leftarrow C\; p \qquad \Psi \vdash \Pi :: \vec{A}\; q \Leftarrow C\; p}{\Psi \vdash (\pi \mid \Pi) :: \vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatchSeq}$$

$$\frac{\Psi \vdash e \Leftarrow C\; p}{\Psi \vdash (\cdot \Rightarrow e) :: \cdot\; q \Leftarrow C\; p}\ \text{DeclMatchBase} \qquad \frac{\Psi \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; q \Leftarrow C\; p}{\Psi \vdash (),\vec{\rho} \Rightarrow e :: 1,\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatchUnit}$$

$$\frac{\Psi, \alpha : \kappa \vdash \vec{\rho} \Rightarrow e :: A,\vec{A}\; q \Leftarrow C\; p}{\Psi \vdash (\vec{\rho} \Rightarrow e) :: (\exists \alpha : \kappa.\, A),\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatch}\exists \qquad \frac{\Psi \vdash \rho_1,\rho_2,\vec{\rho} \Rightarrow e :: A_1,A_2,\vec{A}\; q \Leftarrow C\; p}{\Psi \vdash \langle \rho_1,\rho_2 \rangle,\vec{\rho} \Rightarrow e :: (A_1 \times A_2),\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatch}\times$$

$$\frac{\Psi \vdash \rho,\vec{\rho} \Rightarrow e :: A_k,\vec{A}\; q \Leftarrow C\; p}{\Psi \vdash \mathsf{inj}_k\, \rho,\vec{\rho} \Rightarrow e :: A_1 + A_2,\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatch}+_k$$

$$\frac{\Psi \,/\, P \vdash \vec{\rho} \Rightarrow e :: A,\vec{A}\; ! \Leftarrow C\; p}{\Psi \vdash \vec{\rho} \Rightarrow e :: (A \wedge P),\vec{A}\; ! \Leftarrow C\; p}\ \text{DeclMatch}\wedge \qquad \frac{\Psi \vdash \vec{\rho} \Rightarrow e :: A,\vec{A}\; \not{!} \Leftarrow C\; p}{\Psi \vdash \vec{\rho} \Rightarrow e :: (A \wedge P),\vec{A}\; \not{!} \Leftarrow C\; p}\ \text{DeclMatch}\wedge\not{!}$$

$$\frac{\Psi, \alpha : \mathbb{N} \vdash \rho_1,\rho_2,\vec{\rho} \Rightarrow e :: A, (\mathsf{Vec}\; \alpha\; A),\vec{A}\; \not{!} \Leftarrow C\; p}{\Psi \vdash (\rho_1 :: \rho_2),\vec{\rho} \Rightarrow e :: (\mathsf{Vec}\; t\; A),\vec{A}\; \not{!} \Leftarrow C\; p}\ \text{DeclMatchCons}\not{!}$$

$$\frac{\Psi, \alpha : \mathbb{N} \,/\, (t = \mathsf{succ}(\alpha)) \vdash \rho_1,\rho_2,\vec{\rho} \Rightarrow e :: A, (\mathsf{Vec}\; \alpha\; A),\vec{A}\; ! \Leftarrow C\; p}{\Psi \vdash (\rho_1 :: \rho_2),\vec{\rho} \Rightarrow e :: (\mathsf{Vec}\; t\; A),\vec{A}\; ! \Leftarrow C\; p}\ \text{DeclMatchCons}$$

$$\frac{\Psi \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; \not{!} \Leftarrow C\; p}{\Psi \vdash [\,],\vec{\rho} \Rightarrow e :: (\mathsf{Vec}\; t\; A),\vec{A}\; \not{!} \Leftarrow C\; p}\ \text{DeclMatchNil}\not{!} \qquad \frac{\Psi \,/\, (t = \mathsf{zero}) \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; ! \Leftarrow C\; p}{\Psi \vdash [\,],\vec{\rho} \Rightarrow e :: (\mathsf{Vec}\; t\; A),\vec{A}\; ! \Leftarrow C\; p}\ \text{DeclMatchNil}$$

$$\frac{\begin{array}{l} A \text{ not headed by } \wedge \text{ or } \exists \\ \Psi, x : A ! \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; q \Leftarrow C\; p \end{array}}{\Psi \vdash x,\vec{\rho} \Rightarrow e :: A,\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatchNeg} \qquad \frac{\begin{array}{l} A \text{ not headed by } \wedge \text{ or } \exists \\ \Psi \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; q \Leftarrow C\; p \end{array}}{\Psi \vdash \_,\vec{\rho} \Rightarrow e :: A,\vec{A}\; q \Leftarrow C\; p}\ \text{DeclMatchWild}$$

$\boxed{\Psi \,/\, P \vdash \Pi :: \vec{A}\; ! \Leftarrow C\; p}$ Under context Ψ, incorporate proposition P while checking branches Π with patterns of type $\vec{A}$ and bodies of type C

$$\frac{\mathsf{mgu}(\sigma,\tau) = \bot}{\Psi \,/\, \sigma = \tau \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; ! \Leftarrow C\; p}\ \text{DeclMatch}\bot$$

$$\frac{\mathsf{mgu}(\sigma,\tau) = \theta \qquad \theta(\Psi) \vdash \theta(\vec{\rho} \Rightarrow e) :: \theta(\vec{A})\; q \Leftarrow \theta(C)\; p}{\Psi \,/\, \sigma = \tau \vdash \vec{\rho} \Rightarrow e :: \vec{A}\; ! \Leftarrow C\; p}\ \text{DeclMatchUnify}$$

Fig. 7. Declarative pattern matching

whether a missing nil in a list match should be taken as evidence of coverage failure or that the length is non-zero). As with spine recovery, this rule is only well-founded because the universal quantification ranges over synthesized types over a subterm.

The $\Psi \vdash \Pi :: \vec{A}\; q \Leftarrow C\; p$ judgment (rules in Figure 7) systematically checks the typing of each branch in Π: rule DeclMatchEmpty succeeds on the empty list, and DeclMatchSeq checks one branch and recurs on the remaining branches. Rules for sums, units, and products break down patterns left to right, one constructor at a time. Products also extend the sequences of patterns and types, with DeclMatch$\times$ breaking down a pattern vector headed by a pair pattern $\langle p,p' \rangle,\vec{p}$ into $p,p',\vec{p}$, and breaking down the type sequence $(A \times B),\vec{C}$ into $A,B,\vec{C}$. Once all the patterns are

$\boxed{\begin{array}{l}\Psi \vdash \Pi \; covers \; \vec{A} p \\ \Psi \; / \; P \vdash \Pi \; covers \; \vec{A}! \\ \Pi \text{ guarded}\end{array}}$ Patterns Π cover the types $\vec{A}$ in context Ψ; Patterns Π cover the types $\vec{A}$ in context Ψ, assuming P; Pattern list Π contains a list pattern constructor at the head position

$$\frac{}{\Psi \vdash (\cdot \Rightarrow e_1) \mid \Pi' \; covers \cdot p}\;\text{DeclCoversEmpty} \qquad \frac{\Pi \overset{\text{var}}{\rightsquigarrow} \Pi' \qquad \Psi \vdash \Pi' \; covers \; \vec{A} p}{\Psi \vdash \Pi \; covers \; A, \vec{A} p}\;\text{DeclCoversVar}$$

$$\frac{\Pi \overset{1}{\rightsquigarrow} \Pi' \qquad \Psi \vdash \Pi' \; covers \; \vec{A} p}{\Psi \vdash \Pi \; covers \; 1, \vec{A} p}\;\text{DeclCovers1} \qquad \frac{\Pi \overset{\times}{\rightsquigarrow} \Pi' \qquad \Psi \vdash \Pi' \; covers \; A_1, A_2, \vec{A} p}{\Psi \vdash \Pi \; covers \; (A_1 \times A_2), \vec{A} p}\;\text{DeclCovers}\times$$

$$\frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R \qquad \begin{array}{l}\Psi \vdash \Pi_L \; covers \; A_1, \vec{A} p \\ \Psi \vdash \Pi_R \; covers \; A_2, \vec{A} p\end{array}}{\Psi \vdash \Pi \; covers \; (A_1 + A_2), \vec{A} p}\;\text{DeclCovers+} \qquad \frac{\Psi, \alpha : \kappa \vdash \Pi \; covers \; A, \vec{A} p}{\Psi \vdash \Pi \; covers \; (\exists \alpha : \kappa. \, A), \vec{A} p}\;\text{DeclCovers}\exists$$

$$\frac{\Psi \; / \; t_1 = t_2 \vdash \Pi \; covers \; A_0, \vec{A}!}{\Psi \vdash \Pi \; covers \; \big(A_0 \wedge (t_1 = t_2)\big), \vec{A}!}\;\text{DeclCovers}\wedge \qquad \frac{\Psi \vdash \Pi \; covers \; A_0, \vec{A} \not{!}}{\Psi \vdash \Pi \; covers \; \big(A_0 \wedge (t_1 = t_2)\big), \vec{A} \not{!}}\;\text{DeclCovers}\wedge\not{!}$$

$$\frac{\Pi \text{ guarded} \qquad \Pi \overset{\text{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::} \qquad \begin{array}{l}\Psi \; / \; t = \mathsf{zero} \vdash \Pi_{[]} \; covers \; \vec{A}! \\ \Psi, n : \mathbb{N} \; / \; t = \mathsf{succ}(n) \vdash \Pi_{::} \; covers \; (A, \mathsf{Vec} \; n \; A, \vec{A})!\end{array}}{\Psi \vdash \Pi \; covers \; \mathsf{Vec} \; t \; A, \vec{A}!}\;\text{DeclCoversVec}$$

$$\frac{\Pi \text{ guarded} \qquad \Pi \overset{\text{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::} \qquad \begin{array}{l}\Psi \vdash \Pi_{[]} \; covers \; \vec{A} \not{!} \\ \Psi, n : \mathbb{N} \vdash \Pi_{::} \; covers \; (A, \mathsf{Vec} \; n \; A, \vec{A}) \not{!}\end{array}}{\Psi \vdash \Pi \; covers \; \mathsf{Vec} \; t \; A, \vec{A} \not{!}}\;\text{DeclCoversVec}\not{!}$$

$$\frac{\mathsf{mgu}(t_1, t_2) = \theta \qquad \theta(\Psi) \vdash \theta(\Pi) \; covers \; \theta(\vec{A})\,!}{\Psi \; / \; t_1 = t_2 \vdash \Pi \; covers \; \vec{A}!}\;\text{DeclCoversEq} \qquad \frac{\mathsf{mgu}(t_1, t_2) = \bot}{\Psi \; / \; t_1 = t_2 \vdash \Pi \; covers \; \vec{A}!}\;\text{DeclCoversEqBot}$$

$$\frac{}{[\,], \vec{p} \Rightarrow e \mid \Pi \text{ guarded}} \qquad \frac{}{p :: p', \vec{p} \Rightarrow e \mid \Pi \text{ guarded}} \qquad \frac{\Pi \text{ guarded}}{\_, \vec{p} \Rightarrow e \mid \Pi \text{ guarded}} \qquad \frac{\Pi \text{ guarded}}{x, \vec{p} \Rightarrow e \mid \Pi \text{ guarded}}$$

Fig. 8. Match coverage

eliminated, the DeclMatchBase rule says that if the body typechecks, then the branch typechecks. For completeness, the variable and wildcard rules are restricted so that any top-level existentials and equations are eliminated before discarding the type.

The existential elimination rule DeclMatch$\exists$ unpacks an existential type, and DeclMatch$\wedge$ breaks apart a conjunction by eliminating the equality using unification. The DeclMatch$\bot$ rule says that if the equation is false then typing succeeds, because this case is impossible. The DeclMatchUnify rule unifies the two terms of an equation and applies the substitution before continuing to check typing. Together, these two rules implement the Schroeder-Heister equality elimination rule. Because our language of terms has only simple first-order terms, either unification will fail, or there is a most general unifier. Note, however, that DeclMatch$\wedge$ only applies when the pattern type is principal. Otherwise, we use the DeclMatch$\wedge\not{!}$ rule, which throws away the equation and does not refine any types at all. In this way, we can ensure that we will only try to eliminate equations which are fully known (i.e., principal). Similar considerations apply to vectors, with length information being used to refine types only when the type of the scrutinee is principal.

The $\Psi \vdash \Pi \; covers \; \vec{A} p$ judgment (in Figure 8) checks whether a set of patterns covers all possible cases. As with match typing, we systematically deconstruct the sequence of types in the branch, but

$\boxed{\Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::}}$ Expand vector patterns in Π

$$\frac{}{\cdot \overset{\mathsf{Vec}}{\rightsquigarrow} \cdot \parallel \cdot} \qquad \frac{\rho \in \{x, \_\} \qquad \Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::}}{\left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \left(\vec{\rho} \Rightarrow e\right) \mid \Pi_{[]} \parallel \left(\_, \_, \vec{\rho} \Rightarrow e\right) \mid \Pi_{::}}$$

$$\frac{\Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::}}{\left([], \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \left(\vec{\rho} \Rightarrow e\right) \mid \Pi_{[]} \parallel \Pi_{::}} \qquad \frac{\Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \Pi_{::}}{\left((\rho :: \rho'), \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\mathsf{Vec}}{\rightsquigarrow} \Pi_{[]} \parallel \left(\rho, \rho', \vec{\rho} \Rightarrow e\right) \mid \Pi_{::}}$$

$\boxed{\Pi \overset{\times}{\rightsquigarrow} \Pi'}$ Expand head pair patterns in Π

$$\frac{}{\cdot \overset{\times}{\rightsquigarrow} \cdot} \qquad \frac{\Pi \overset{\times}{\rightsquigarrow} \Pi'}{\left(\langle \rho_1, \rho_2 \rangle, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\times}{\rightsquigarrow} \left(\rho_1, \rho_2, \vec{\rho} \Rightarrow e\right) \mid \Pi'} \qquad \frac{\rho \in \{z, \_\} \qquad \Pi \overset{\times}{\rightsquigarrow} \Pi'}{\left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\times}{\rightsquigarrow} \left(\_, \_, \vec{\rho} \Rightarrow e\right) \mid \Pi'}$$

$\boxed{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}$ Expand head sum patterns in Π into left Π_L and right Π_R sets

$$\frac{}{\cdot \overset{+}{\rightsquigarrow} \cdot \parallel \cdot} \qquad \frac{\rho \in \{x, \_\} \qquad \Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{\left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{+}{\rightsquigarrow} \left(\_, \vec{\rho} \Rightarrow e\right) \mid \Pi_L \parallel \left(\_, \vec{\rho} \Rightarrow e\right) \mid \Pi_R}$$

$$\frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{\left(\mathsf{inj}_1\, \rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{+}{\rightsquigarrow} \left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi_L \parallel \Pi_R} \qquad \frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{\left(\mathsf{inj}_2\, \rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi_R}$$

$\boxed{\Pi \overset{\mathrm{var}}{\rightsquigarrow} \Pi'}$ Remove head variable and wildcard patterns from Π

$\boxed{\Pi \overset{1}{\rightsquigarrow} \Pi'}$ Remove head variable, wildcard, and unit patterns from Π

$$\frac{}{\cdot \overset{\mathrm{var}}{\rightsquigarrow} \cdot} \qquad \frac{\rho \in \{x, \_\} \qquad \Pi \overset{\mathrm{var}}{\rightsquigarrow} \Pi'}{\left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{\mathrm{var}}{\rightsquigarrow} \left(\vec{\rho} \Rightarrow e\right) \mid \Pi'} \qquad \frac{}{\cdot \overset{1}{\rightsquigarrow} \cdot} \qquad \frac{\rho \in \{x, \_, ()\} \qquad \Pi \overset{1}{\rightsquigarrow} \Pi'}{\left(\rho, \vec{\rho} \Rightarrow e\right) \mid \Pi \overset{1}{\rightsquigarrow} \left(\vec{\rho} \Rightarrow e\right) \mid \Pi'}$$

Fig. 9. Pattern expansion

we also need auxiliary operations to *expand* the patterns. For example, the $\Pi \overset{\times}{\rightsquigarrow} \Pi'$ operation takes every branch $\langle p, p' \rangle, \vec{\rho} \Rightarrow e$ and expands it to $p, p', \vec{\rho} \Rightarrow e$. To keep the sequence of patterns aligned with the sequence of types, we also expand variables and wildcard patterns into two wildcards: $x, \vec{\rho} \Rightarrow e$ becomes $\_, \_, \vec{\rho} \Rightarrow e$. After expanding out all the pairs, DeclCovers× checks coverage by breaking down the pair type.

For sum types, we expand a list of branches into *two* lists, one for each injection. So $\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R$ will send all branches headed by $\mathsf{inj}_1\, p$ into Π_L and all branches headed by $\mathsf{inj}_2\, p$ into Π_R, with variables and wildcards being sent to both sides. Then DeclCovers+ checks the left and right branches independently.

As with typing, DeclCovers∃ just unpacks the existential type. Likewise, DeclCoversEqBot and DeclCoversEq handle the two cases arising from equations. If an equation is unsatisfiable, coverage succeeds since there are no possible values of that type. If it is satisfiable, we apply the substitution and continue coverage checking. Just as when typechecking patterns, we only use property types to refine coverage checking when the equations come from a principal type — the DeclCovers∧$\not!$ rule simply throws away the equation when the type is not principal. (This is a sound approximation which ends up requiring more patterns when the type is not principal.)

So far, the coverage rules for pattern matching are almost purely type-directed. However, once recursive types like $\mathsf{Vec}\ n\ A$ enter the picture, matters become a little more subtle. The issue is that if we split a wildcard _ of type $\mathsf{Vec}\ n\ A$, the type *doesn't tell us when to stop*. That is, we could split a wildcard into a nil $[]$ and cons $\_ :: \_$ pattern; or we could turn it into a nil $[]$, singleton $\_ :: []$

and two-or-longer _ :: _ :: _ pattern; and so on. The key issue is that the tail of a list has the same set of possible patterns as the list itself, and so blindly following the type structure will not ensure termination of coverage checking.

In this paper, we take the view that the patterns the programmer wrote should guide how much to split types when doing coverage checking for inductive types. In Figure 8, we introduce the Π guarded judgement, which checks to see if a constructor pattern is present in the leading column of patterns. If it is, then our algorithm will unfold the recursive type as part of type checking, and otherwise it will not. This is by no means a canonical choice: our choice is similar to the choice Agda makes, but other language implementations make other choices. In contrast, the OCaml coverage checking algorithm unfolds wildcard patterns at GADT type one step more than what the programmer wrote [Garrigue and Le Normand 2015]. (They also observe that precise exhaustiveness checking is undecidable, meaning that some choice of heuristic is unavoidable.)

4.3 Design Considerations for Pattern Matching

Evaluation Order. Our typing and coverage checking rules are given assuming a *call-by-value* evaluation strategy. These coverage rules are not sound under a call-by-name evaluation order. Consider the following program, writing $\bot$ for a looping term:

$$\mathsf{case}(\bot : A \wedge (s = t), x \Rightarrow e)$$

When type-checking this program, the DeclMatch$\wedge$ and DeclCovers$\wedge$ rules are permitted to eliminate the equality $s = t$ when checking e. However, one can use a looping program to inhabit $A \wedge (s = t)$ for any P, and so we have introduced a spurious equality into the context when checking e. In contrast, in a call-by-value language the scrutinee of a case will be reduced before the match proceeds, so this issue cannot arise. (In a total language such as Koka, these rules would be sound irrespective of evaluation order, since all evaluation strategies are indistinguishable.)

Redundant Patterns. These rules do not check for redundancy: DeclCoversEmpty applies even when branches are left over. When DeclCoversEmpty is applied, we could mark the $\cdot \Rightarrow e_1$ branch, and issue a warning for unmarked branches. This seems better as a warning than an error, since redundancy is not stable under substitution. For example, a case over (Vec n A) with [] and :: branches is not redundant—but if we substitute 0 for n, the :: branch becomes redundant.

Synthesis. Bidirectional typing is a form of partial type inference, which Pierce and Turner [2000] said should "eliminate especially those type annotations that are both *common* and *silly*". But our rules are rather parsimonious in what they synthesize; for instance, () does not synthesize 1, and so might need an annotation. Fortunately, it would be straightforward to add such rules, following the style of Dunfield and Krishnaswami [2013].

5 ALGORITHMIC TYPING

Our algorithmic rules closely mimic our declarative rules, except that whenever a declarative rule would make a guess, the algorithmic rule adds to the context an existential variable (written with a hat $\hat{\alpha}$). As typechecking proceeds, we add solutions to the existential variables, reflecting increasing knowledge. Hence, each declarative typing judgment has a corresponding algorithmic judgment with an output context as well as an input context. The algorithmic type checking judgment $\Gamma \vdash e \Leftarrow A\ p \dashv \Delta$ takes an input context Γ and yields an output context Δ that includes increased knowledge about what the types have to be. The notion of increasing knowledge is formalized by a judgment $\Gamma \longrightarrow \Delta$ (Section 5.3).

Figure 10 shows a dependency graph of the algorithmic judgments. Each declarative judgment has a corresponding algorithmic judgment, but the algorithmic system adds judgments such as type

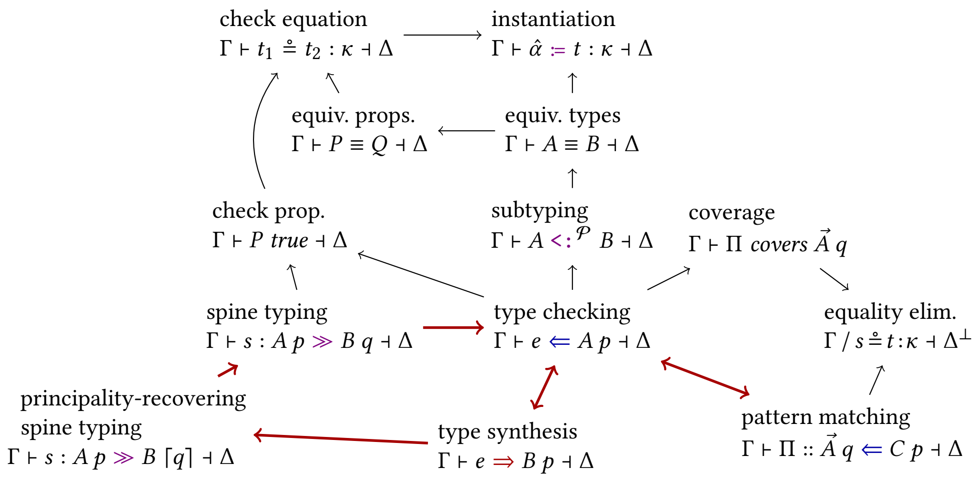

Fig. 10. Dependency structure of the algorithmic judgments

equivalence checking $\Gamma \vdash A \equiv B \dashv \Delta$ and variable instantiation $\Gamma \vdash \hat{\alpha} := t : \kappa \dashv \Delta$. Declaratively, these judgments correspond to uses of reflexivity axioms; algorithmically, they correspond to solving existential variables to equate terms.

We give the algorithmic typing rules in Figure 14; rules for most other judgments are in the appendix. Our style of specification broadly follows Dunfield and Krishnaswami [2013]: we adapt their mechanisms of variable instantiation, context extension, and context application (to both types and other contexts). Our versions of these mechanisms, however, support indices, equations over universal variables, and the $\exists/\supset/\wedge$ connectives. We also differ in our formulation of spine typing, and by being able to track which types are principal.

5.1 Examples

To show how the spine typing rules recover principality, we present some example derivations.

Suppose we have an identity function *id*, defined in an algorithmic context Γ by the hypothesis $\mathit{id} : (\forall \alpha : \star.\ \alpha \to \alpha)\,!$. Since the hypothesis has $!$, the type of *id* is known to be principal. If we apply *id* to $()$, we expect to get something of unit type 1. Despite the $\forall$ in the type of *id*, the resulting type should be principal, because no other type is possible. We can indeed derive that type:

$$\dfrac{\dfrac{(\mathit{id} : (\forall \alpha : \star.\ \alpha \to \alpha)\,!) \in \Gamma}{\Gamma \vdash \mathit{id} \Rightarrow (\forall \alpha : \star.\ \alpha \to \alpha)\,! \dashv \Gamma}\ \text{Var} \qquad \Gamma \vdash ((\,)\ \cdot) : (\forall \alpha : \star.\ \alpha \to \alpha)\,! \gg 1\ \lceil ! \rceil \dashv \Gamma, \hat{\alpha} : \star = 1}{\Gamma \vdash \mathit{id}\ ((\,)\ \cdot) \Rightarrow 1\,! \dashv \Gamma, \hat{\alpha} : \star = 1}\ {\to}\mathrm{E}$$

(Here, we write the application $\mathit{id}\ ()$ as $\mathit{id}\ ((\,)\ \cdot)$, to show the structure of the spine as analyzed by the typing rules.) In the derivation of the second premise of $\to$E, shown below, we can follow the evolution of the principality marker.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\hat\alpha{:}\star \vdash () \Leftarrow \hat\alpha\ \not{!} \dashv \Gamma,\hat\alpha:\star=1}\,1\mathsf{I}\hat\alpha \quad \dfrac{}{\Gamma,\hat\alpha:\star=1 \vdash \cdot : 1\ \not{!} \gg 1\ \not{!} \dashv \Gamma,\hat\alpha:\star=1}\,\textsf{EmptySpine}}{\Gamma,\hat\alpha{:}\star \vdash (()\ \cdot) : \hat\alpha\to\hat\alpha\ \not{!} \gg 1\ \not{!} \dashv \Gamma,\hat\alpha:\star=1}\,{\to}\textsf{Spine}}{\Gamma \vdash (()\ \cdot) : (\forall\alpha:\star.\,\alpha\to\alpha)\ ! \gg 1\ \not{!} \dashv \Gamma,\hat\alpha:\star=1}\,\forall\textsf{Spine} \quad \mathsf{FEV}(1)=\emptyset}{\Gamma \vdash (()\ \cdot) : (\forall\alpha:\star.\,\alpha\to\alpha)\ \underbrace{!}_{\text{input}} \gg 1\ \lceil ! \rceil \dashv \Gamma,\hat\alpha:\star=1}\,\textsf{SpineRecover}$$

- The input principality (marked "input") is $!$, because the input type $(\forall\alpha:\star.\ \alpha\to\alpha)$ was marked as principal in the hypothesis typing *id*.
- Rule SpineRecover begins by invoking the ordinary (non-recovering) spine judgment, passing all inputs unchanged, including the principality $!$.
- Rule $\forall$Spine adds an existential variable $\hat\alpha$ to represent the instantiation of the quantified type variable α, and substitutes $\hat\alpha$ for α. Since this instantiation is, in general, *not* principal, it replaces $!$ with $\not{!}$ (highlighted) in its premise. This marks the type $\hat\alpha\to\hat\alpha$ as non-principal.
- Rule $\to$Spine decomposes $\hat\alpha\to\hat\alpha$ and checks $()$ against $\hat\alpha$, maintaining the principality $\not{!}$. Once principality is lost, it can only be recovered within the SpineRecover rule itself.
- Rule $1\mathsf{I}\hat\alpha$ notices that we are checking $()$ against an unknown type $\hat\alpha$; since the expression is $()$, the type $\hat\alpha$ must be 1, so it adds that solution to its output context.
- Moving to the second premise of $\to$Spine, we analyze the remaining part of the spine. That is just the empty spine $\cdot$, and rule EmptySpine passes its inputs along as outputs. In particular, the principality $\not{!}$ is unchanged.
- The principalities are passed down to the conclusion of $\forall$Spine, where $\not{!}$ is highlighted.
- In SpineRecover, we notice that the output type 1 has no existential variables ($\mathsf{FEV}(1)=\emptyset$), which allows us to recover principality of the output type: $\lceil ! \rceil$.

In the corresponding derivation in our declarative system, we have, instead, a check that no other types are derivable:

$$\dfrac{\dfrac{\Psi\vdash 1:\star \quad \dfrac{\dfrac{}{\Psi\vdash () \Leftarrow 1\ \not{!}}\,\textsf{Decl1I} \quad \dfrac{}{\Psi\vdash \cdot : 1\ \not{!}\gg 1\ \not{!}}\,\textsf{DeclEmptySpine}}{\Psi\vdash (()\ \cdot) : 1\to 1\ \not{!}\gg 1\ \not{!}}\,\textsf{Decl}{\to}\textsf{Spine}}{\Psi\vdash(()\ \cdot):(\forall\alpha:\star.\,\alpha\to\alpha)\ !\gg 1\ \not{!}}\,\textsf{Decl}\forall\textsf{Spine} \quad \begin{array}{l}\text{for all } C'.\ \text{ if}\\ \Psi\vdash(()\ \cdot)\\ \quad :(\forall\alpha{:}\star.\,\alpha\to\alpha)\ !\gg C'\ \not{!}\\ \text{then } C'=1\end{array}}{\Psi\vdash(()\ \cdot):(\forall\alpha:\star.\,\alpha\to\alpha)\ \underbrace{!}_{\text{input}}\gg 1\ \lceil ! \rceil}\,\textsf{DeclSpineRecover}$$

Here, we highlight the replacement in Decl$\forall$Spine of the quantified type variable α by the "guessed" solution 1. The second premise of DeclSpineRecover checks that no other output type C' could have been produced, no matter what solution was chosen by Decl$\forall$Spine for α.

Syntax. Expressions are the same as in the declarative system.

Existential variables. The algorithmic system adds existential variables $\hat\alpha$, $\hat\beta$, $\hat\gamma$ to types and terms/monotypes (Figure 11). We use the same meta-variables A, ... We write u for either a universal variable α or an existential variable $\hat\alpha$.

Contexts. An algorithmic context Γ is a sequence that, like a declarative context, may contain universal variable declarations $\alpha : \kappa$ and expression variable typings $x : A\,p$. However, it may also have (1) *unsolved* existential variable declarations $\hat\alpha : \kappa$ (included in the $\Gamma, u : \kappa$ production); (2) *solved* existential variable declarations $\hat\alpha : \kappa = \tau$; (3) *equations* over universal variables $\alpha = \tau$; and

$$
\begin{array}{lrcl}
\text{Universal variables} & \alpha, \beta, \gamma & & \\
\text{Existential variables} & \hat{\alpha}, \hat{\beta}, \hat{\gamma} & & \\
\text{Variables} & u & ::= & \alpha \mid \hat{\alpha} \\
\text{Types} & A, B, C & ::= & 1 \mid A \to B \mid A + B \mid A \times B \mid \alpha \mid \hat{\alpha} \mid \forall \alpha : \kappa.\, A \mid \exists \alpha : \kappa.\, A \\
& & & \mid P \supset A \mid A \land P \mid \mathsf{Vec}\ t\ A \\
\text{Propositions} & P, Q & ::= & t = t' \\
\text{Binary connectives} & \oplus & ::= & \to \mid + \mid \times \\
\text{Terms/monotypes} & t, \tau, \sigma & ::= & \mathsf{zero} \mid \mathsf{succ}(t) \mid 1 \mid \alpha \mid \hat{\alpha} \mid \tau \to \sigma \mid \tau + \sigma \mid \tau \times \sigma \\
\text{Contexts} & \Gamma, \Delta, \Theta & ::= & \cdot \mid \Gamma, u : \kappa \mid \Gamma, x : A p \mid \Gamma, \hat{\alpha} : \kappa = \tau \mid \Gamma, \alpha = t \mid \Gamma, \blacktriangleright_u \\
\text{Complete contexts} & \Omega & ::= & \cdot \mid \Omega, \alpha : \kappa \mid \Omega, x : A p \mid \Omega, \hat{\alpha} : \kappa = \tau \mid \Omega, \alpha = t \mid \Omega, \blacktriangleright_u
\end{array}
$$

$$\text{Possibly inconsistent contexts} \quad \Delta^{\perp} \ ::= \ \Delta \mid \perp$$

Fig. 11. Syntax of types, contexts, and other objects in the algorithmic system

$$
\begin{array}{lcl}
[\Gamma]\alpha & = & \begin{cases} [\Gamma]\tau \text{ when } (\alpha = \tau) \in \Gamma \\ \alpha \quad \text{otherwise} \end{cases} \\
[\Gamma](P \supset A) & = & ([\Gamma]P) \supset ([\Gamma]A) \\
[\Gamma](A \land P) & = & ([\Gamma]A) \land ([\Gamma]P) \\
[\Gamma](A \oplus B) & = & ([\Gamma]A) \oplus ([\Gamma]B) \\
[\Gamma](\mathsf{Vec}\ t\ A) & = & \mathsf{Vec}\ ([\Gamma]t)\ ([\Gamma]A)
\end{array}
\qquad
\begin{array}{lcl}
\big[\Gamma[\hat{\alpha} : \kappa = \tau]\big]\hat{\alpha} & = & [\Gamma]\tau \\
\big[\Gamma[\hat{\alpha} : \kappa]\big]\hat{\alpha} & = & \hat{\alpha} \\
[\Gamma](\forall \alpha : \kappa.\, A) & = & \forall \alpha : \kappa.\, [\Gamma]A \\
[\Gamma](\exists \alpha : \kappa.\, A) & = & \exists \alpha : \kappa.\, [\Gamma]A \\
[\Gamma](t_1 = t_2) & = & ([\Gamma]t_1) = ([\Gamma]t_2)
\end{array}
$$

Fig. 12. Applying a context, as a substitution, to a type

(4) *markers* $\blacktriangleright_u$. An equation $\alpha = \tau$ must appear to the right of the universal variable's declaration $\alpha : \kappa$. We use markers as delimiters within contexts. For example, rule $\supset$I adds $\blacktriangleright_P$, which tells it how much of its last premise's output context $(\Delta, \blacktriangleright_P, \Delta')$ should be dropped. (We abuse notation by writing $\blacktriangleright_P$ rather than cluttering the context with a dummy α and writing $\blacktriangleright_\alpha$.)

A complete algorithmic context, denoted by Ω, is an algorithmic context with no unsolved existential variable declarations.

Assuming an equality can yield inconsistency: for example, $\mathsf{zero} = \mathsf{succ}(\mathsf{zero})$. We write $\Delta^{\perp}$ for either a valid algorithmic context Δ or inconsistency $\perp$.

5.2 Context Substitution $[\Gamma]A$ and Hole Notation $\Gamma[\Theta]$

An algorithmic context can be viewed as a substitution for its solved existential variables. For example, $\hat{\alpha} = 1, \hat{\beta} = \hat{\alpha} \to 1$ can be applied as if it were the substitution $1/\hat{\alpha}, (\hat{\alpha} \to 1)/\hat{\beta}$ (applied right to left), or the simultaneous substitution $1/\hat{\alpha}, (1 \to 1)/\hat{\beta}$. We write $[\Gamma]A$ for Γ applied as a substitution (Figure 12).

Applying a complete context to a type A (provided it is well-formed: $\Omega \vdash A$ *type*) yields a type $[\Omega]A$ with no existentials. Such a type is well-formed under the *declarative* context obtained by dropping all the existential declarations and applying Ω to declarations $x : A$ (to yield $x : [\Omega]A$). We can think of this context as the result of applying Ω to itself: $[\Omega]\Omega$. More generally, we can apply Ω to any context Γ that it extends: context application $[\Omega]\Gamma$ is given in Figure 13. The application $[\Omega]\Gamma$ is defined if and only if $\Gamma \longrightarrow \Omega$ (context extension; see Section 5.3), and applying Ω to any such Γ yields the same declarative context $[\Omega]\Omega$.

$$
\begin{array}{lcl}
[\cdot]\cdot & = & \cdot \\
[\Omega, x : A\,p](\Gamma, x : A_\Gamma\, p) & = & [\Omega]\Gamma,\ x : [\Omega]A\,p \ \text{ if } [\Omega]A = [\Omega]A_\Gamma \\
[\Omega, \alpha : \kappa](\Gamma, \alpha : \kappa) & = & [\Omega]\Gamma,\ \alpha : \kappa \\
[\Omega, \blacktriangleright_u](\Gamma, \blacktriangleright_u) & = & [\Omega]\Gamma \\
[\Omega, \alpha = t](\Gamma, \alpha = t') & = & \big[[\Omega]t/\alpha\big][\Omega]\Gamma \quad \text{if } [\Omega]t = [\Omega]t' \\
[\Omega, \hat{\alpha} : \kappa = t]\Gamma & = & \begin{cases} [\Omega]\Gamma' \text{ when } \Gamma = (\Gamma', \hat{\alpha} : \kappa = t') \\ [\Omega]\Gamma' \text{ when } \Gamma = (\Gamma', \hat{\alpha} : \kappa) \\ [\Omega]\Gamma \ \text{ otherwise} \end{cases}
\end{array}
$$

Fig. 13. Applying a complete context Ω to a context

In addition to appending declarations (as in the declarative system), we sometimes insert and replace declarations, so a notation for contexts with a hole is useful: $\Gamma = \Gamma_0[\Theta]$ means Γ has the form $(\Gamma_L, \Theta, \Gamma_R)$. For example, if $\Gamma = \Gamma_0[\hat{\beta}] = (\hat{\alpha}, \hat{\beta}, x : \hat{\beta})$, then $\Gamma_0[\hat{\beta} = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta})$.

We also use contexts with *two* ordered holes: if $\Gamma = \Gamma_0[\Theta_1][\Theta_2]$ then $\Gamma = (\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R)$.

5.3 The Context Extension Relation $\Gamma \longrightarrow \Delta$

A context Γ *is extended by* a context Δ, written $\Gamma \longrightarrow \Delta$, if Δ has at least as much information as Γ, while conforming to the same declarative context—that is, $[\Omega]\Gamma = [\Omega]\Delta$ for some Ω. In a sense, $\Gamma \longrightarrow \Delta$ says that Γ is *entailed by* Δ: all positive information derivable from Γ can also be derived from Δ (which may have more information, say, that $\hat{\alpha}$ is equal to a particular type). We give the rules for extension in Figure 15.

The rules deriving the context extension judgment (Figure 15) say that the empty context extends the empty context ($\longrightarrow$Id); a term variable typing with A' extends one with A if applying the extending context Δ to A and A' yields the same type ($\longrightarrow$Var); universal variable declarations and equations must match ($\longrightarrow$Uvar, $\longrightarrow$Eqn); scope markers must match ($\longrightarrow$Marker); and, existential variables may either match ($\longrightarrow$Unsolved, $\longrightarrow$Solved), get solved by the extending context ($\longrightarrow$Solve), or be added by the extending context ($\longrightarrow$Add, $\longrightarrow$AddSolved).

Extension may change solutions, if information is preserved or increased: $(\hat{\alpha} : \star, \hat{\beta} : \star = \hat{\alpha}) \longrightarrow (\hat{\alpha} : \star = 1, \hat{\beta} : \star = \hat{\alpha})$ directly increases information about $\hat{\alpha}$, and indirectly increases information about $\hat{\beta}$. More interestingly, if $\Delta = (\hat{\alpha}{:}\star = 1, \hat{\beta}{:}\star = \hat{\alpha})$ and $\Omega = (\hat{\alpha}{:}\star = 1, \hat{\beta}{:}\star = 1)$, then $\Delta \longrightarrow \Omega$: while the solution of $\hat{\beta}$ in Ω is different, in the sense that Ω contains $\hat{\beta} : \star = 1$ while Δ contains $\hat{\beta} : \star = \hat{\alpha}$, applying Ω to the solutions gives the same result: $[\Omega]\hat{\alpha} = [\Omega]1 = 1$, the same as $[\Omega]1 = 1$.

Extension is quite rigid, however, in two senses. First, if a declaration appears in Γ, it appears in all extensions of Γ. Second, *extension preserves order*. For example, if $\hat{\beta}$ is declared after $\hat{\alpha}$ in Γ, then $\hat{\beta}$ will also be declared after $\hat{\alpha}$ in every extension of Γ. This holds for every variety of declaration, including equations of universal variables. This rigidity aids in enforcing type variable scoping and dependencies, which are nontrivial in a setting with higher-rank polymorphism.

5.4 Determinacy

Given appropriate inputs (Γ, e, A, p) to the algorithmic judgments, only one set of outputs (C, q, Δ) is derivable (Theorem 5 in the supplementary material, p. 30). We use this property (for spine judgments) in the proof of soundness.

$\Gamma \vdash e \Leftarrow A\,p \dashv \Delta$	Under input context Γ, expression e checks against input type A, with output context Δ
$\Gamma \vdash e \Rightarrow A\,p \dashv \Delta$	Under input context Γ, expression e synthesizes output type A, with output context Δ

$$\frac{(x : A\,p) \in \Gamma}{\Gamma \vdash x \Rightarrow [\Gamma]A\,p \dashv \Gamma}\;\textsf{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A\,q \dashv \Theta \qquad \Theta \vdash A <:^{\mathrm{join}(\mathrm{pol}(B),\mathrm{pol}(A))} B \dashv \Delta}{\Gamma \vdash e \Leftarrow B\,p \dashv \Delta}\;\textsf{Sub}$$

$$\frac{\Gamma \vdash A\,!\;\mathit{type} \qquad \Gamma \vdash e \Leftarrow [\Gamma]A\,! \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow [\Delta]A\,! \dashv \Delta}\;\textsf{Anno} \qquad \frac{\Gamma, x : A\,p \vdash v \Leftarrow A\,p \dashv \Delta, x : A\,p, \Theta}{\Gamma \vdash \mathsf{rec}\;x.\,v \Leftarrow A\,p \dashv \Delta}\;\textsf{Rec}$$

$$\frac{}{\Gamma \vdash () \Leftarrow 1\,p \dashv \Gamma}\;\textsf{1I} \qquad \frac{}{\Gamma[\hat\alpha : \star] \vdash () \Leftarrow \hat\alpha \dashv \Gamma[\hat\alpha : \star = 1]}\;\textsf{1I}\hat\alpha$$

$$\frac{v\;\mathit{chk\text{-}I} \qquad \Gamma, \alpha : \kappa \vdash v \Leftarrow A\,p \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash v \Leftarrow \forall \alpha : \kappa.\,A\,p \dashv \Delta}\;\forall\textsf{I} \qquad \frac{e\;\mathit{chk\text{-}I} \qquad \Gamma, \hat\alpha : \kappa \vdash e \Leftarrow [\hat\alpha/\alpha]A \dashv \Delta}{\Gamma \vdash e \Leftarrow \exists \alpha : \kappa.\,A\,p \dashv \Delta}\;\exists\textsf{I}$$

$$\frac{\begin{array}{l} v\;\mathit{chk\text{-}I} \\ \Gamma, \blacktriangleright_P \,/\, P \dashv \Theta \\ \Theta \vdash v \Leftarrow [\Theta]A\,! \dashv \Delta, \blacktriangleright_P, \Delta' \end{array}}{\Gamma \vdash v \Leftarrow P \supset A\,! \dashv \Delta}\;\supset\textsf{I} \qquad \frac{v\;\mathit{chk\text{-}I} \qquad \Gamma, \blacktriangleright_P \,/\, P \dashv \bot}{\Gamma \vdash v \Leftarrow P \supset A\,! \dashv \Gamma}\;\supset\textsf{I}\bot \qquad \frac{\begin{array}{l} e\text{ not a case} \\ \Gamma \vdash P\;\mathit{true} \dashv \Theta \\ \Theta \vdash e \Leftarrow [\Theta]A\,p \dashv \Delta \end{array}}{\Gamma \vdash e \Leftarrow A \wedge P\,p \dashv \Delta}\;\wedge\textsf{I}$$

$$\frac{\Gamma, x : A\,p \vdash e \Leftarrow B\,p \dashv \Delta, x : A\,p, \Theta}{\Gamma \vdash \lambda x.\,e \Leftarrow A \to B\,p \dashv \Delta}\;\to\textsf{I} \qquad \frac{\Gamma[\hat\alpha_1{:}\star, \hat\alpha_2{:}\star, \hat\alpha{:}\star = \hat\alpha_1 {\to} \hat\alpha_2], x : \hat\alpha_1 \vdash e \Leftarrow \hat\alpha_2 \dashv \Delta, x : \hat\alpha_1, \Delta'}{\Gamma[\hat\alpha : \star] \vdash \lambda x.\,e \Leftarrow \hat\alpha \dashv \Delta}\;\to\textsf{I}\hat\alpha$$

$$\frac{\begin{array}{l} \Gamma \vdash e \Rightarrow A\,p \dashv \Theta \\ \Theta \vdash s : A\,p \gg C\,\lceil q \rceil \dashv \Delta \end{array}}{\Gamma \vdash e\,s \Rightarrow C\,q \dashv \Delta}\;\to\textsf{E} \qquad \frac{\Gamma \vdash e \Rightarrow A\,q \dashv \Theta \qquad \begin{array}{l} \Theta \vdash \Pi :: [\Theta]A\,q \Leftarrow [\Theta]C\,p \dashv \Delta \\ \Delta \vdash \Pi\;\mathit{covers}\;[\Delta]A\,q \end{array}}{\Gamma \vdash \mathsf{case}(e, \Pi) \Leftarrow C\,p \dashv \Delta}\;\textsf{Case}$$

$\Gamma \vdash s : A\,p \gg C\,q \dashv \Delta$ $\Gamma \vdash s : A\,p \gg C\,\lceil q \rceil \dashv \Delta$	Under input context Γ, passing spine s to a function of type A synthesizes type C; in the $\lceil q \rceil$ form, recover principality in q if possible

$$\frac{\Gamma, \hat\alpha : \kappa \vdash e\,s : [\hat\alpha/\alpha]A \gg C\,q \dashv \Delta}{\Gamma \vdash e\,s : \forall \alpha : \kappa.\,A\,p \gg C\,q \dashv \Delta}\;\forall\textsf{Spine} \qquad \frac{\Gamma \vdash P\;\mathit{true} \dashv \Theta \qquad \Theta \vdash e\,s : [\Theta]A\,p \gg C\,q \dashv \Delta}{\Gamma \vdash e\,s : P \supset A\,p \gg C\,q \dashv \Delta}\;\supset\textsf{Spine}$$

$$\frac{}{\Gamma \vdash \cdot : A\,p \gg A\,p \dashv \Gamma}\;\textsf{EmptySpine} \qquad \frac{\Gamma \vdash e \Leftarrow A\,p \dashv \Theta \qquad \Theta \vdash s : [\Theta]B\,p \gg C\,q \dashv \Delta s}{\Gamma \vdash e\,s : A \to B\,p \gg C\,q \dashv \Delta}\;\to\textsf{Spine}$$

$$\frac{\Gamma[\hat\alpha_2{:}\star, \hat\alpha_1{:}\star, \hat\alpha{:}\star = \hat\alpha_1 {\to} \hat\alpha_2] \vdash e\,s : (\hat\alpha_1 \to \hat\alpha_2) \gg C \dashv \Delta}{\Gamma[\hat\alpha : \star] \vdash e\,s : \hat\alpha \gg C \dashv \Delta}\;\hat\alpha\textsf{Spine}$$

$$\frac{\begin{array}{l} \Gamma \vdash s : A\,! \gg C\,\not! \dashv \Delta \\ \mathsf{FEV}(C) = \emptyset \end{array}}{\Gamma \vdash s : A\,! \gg C\,\lceil ! \rceil \dashv \Delta}\;\textsf{SpineRecover} \qquad \frac{\begin{array}{l} \Gamma \vdash s : A\,p \gg C\,q \dashv \Delta \\ \big((p = \not!) \text{ or } (q = !) \text{ or } (\mathsf{FEV}(C) \neq \emptyset)\big) \end{array}}{\Gamma \vdash s : A\,p \gg C\,\lceil q \rceil \dashv \Delta}\;\textsf{SpinePass}$$

Fig. 14. Algorithmic typing, omitting rules for $\times$, $+$, and Vec

$\boxed{\Gamma \longrightarrow \Delta}$ Γ is extended by Δ

$$\frac{}{\cdot \longrightarrow \cdot}\longrightarrow\mathsf{Id} \qquad \frac{\Gamma \longrightarrow \Delta \qquad [\Delta]A = [\Delta]A'}{\Gamma, x : A\, p \longrightarrow \Delta, x : A'\, p}\longrightarrow\mathsf{Var} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \alpha : \kappa \longrightarrow \Delta, \alpha : \kappa}\longrightarrow\mathsf{Uvar}$$

$$\frac{\Gamma \longrightarrow \Delta \qquad [\Delta]t = [\Delta]t'}{\Gamma, \alpha = t \longrightarrow \Delta, \alpha = t'}\longrightarrow\mathsf{Eqn} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\alpha} : \kappa \longrightarrow \Delta, \hat{\alpha} : \kappa}\longrightarrow\mathsf{Unsolved} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_u \longrightarrow \Delta, \blacktriangleright_u}\longrightarrow\mathsf{Marker}$$

$$\frac{\Gamma \longrightarrow \Delta \qquad [\Delta]t = [\Delta]t'}{\Gamma, \hat{\alpha} : \kappa = t \longrightarrow \Delta, \hat{\alpha} : \kappa = t'}\longrightarrow\mathsf{Solved} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma, \hat{\beta} : \kappa' \longrightarrow \Delta, \hat{\beta} : \kappa' = t}\longrightarrow\mathsf{Solve}$$

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha} : \kappa}\longrightarrow\mathsf{Add} \qquad \frac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat{\alpha} : \kappa = t}\longrightarrow\mathsf{AddSolved}$$

Fig. 15. Context extension

6 SOUNDNESS

We show that the algorithmic system is sound with respect to the declarative system. Soundness for the mutually recursive judgments depends on lemmas for the auxiliary judgments (instantiation, equality elimination, checkprop, algorithmic subtyping and match coverage), which are in Appendix J for space reasons. The main soundness result has mutually recursive parts for checking, synthesis, spines and matching—including the principality-recovering spine judgment.

Theorem 6.8 (Soundness of Algorithmic Typing). *Given* $\Delta \longrightarrow \Omega$*:*

(i) *If* $\Gamma \vdash e \Leftarrow A\, p \dashv \Delta$ *and* $\Gamma \vdash A\, p$ *type then* $[\Omega]\Delta \vdash [\Omega]e \Leftarrow [\Omega]A\, p$.
(ii) *If* $\Gamma \vdash e \Rightarrow A\, p \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]e \Rightarrow [\Omega]A\, p$.
(iii) *If* $\Gamma \vdash s : A\, p \gg B\, q \dashv \Delta$ *and* $\Gamma \vdash A\, p$ *type then* $[\Omega]\Delta \vdash [\Omega]s : [\Omega]A\, p \gg [\Omega]B\, q$.
(iv) *If* $\Gamma \vdash s : A\, p \gg B\, \lceil q \rceil \dashv \Delta$ *and* $\Gamma \vdash A\, p$ *type then* $[\Omega]\Delta \vdash [\Omega]s : [\Omega]A\, p \gg [\Omega]B\, \lceil q \rceil$.
(v) *If* $\Gamma \vdash \Pi :: \vec{A}\, q \Leftarrow C\, p \dashv \Delta$ *and* $\Gamma \vdash \vec{A}\, q$ *types and* $[\Gamma]\vec{A} = \vec{A}$ *and* $\Gamma \vdash C\, p$ *type then* $[\Omega]\Delta \vdash [\Omega]\Pi :: [\Omega]\vec{A}\, q \Leftarrow [\Omega]C\, p$.
(vi) *If* $\Gamma \,/\, P \vdash \Pi :: \vec{A}\,! \Leftarrow C\, p \dashv \Delta$ *and* $\Gamma \vdash P$ *prop and* $\mathsf{FEV}(P) = \emptyset$ *and* $[\Gamma]P = P$ *and* $\Gamma \vdash \vec{A}\,!$ *types and* $\Gamma \vdash C\, p$ *type then* $[\Omega]\Delta \,/\, [\Omega]P \vdash [\Omega]\Pi :: [\Omega]\vec{A}\,! \Leftarrow [\Omega]C\, p$.

Much of this proof "turns the crank": apply the induction hypothesis to each premise, yielding derivations of corresponding declarative judgments (with Ω applied everywhere), then apply the corresponding declarative rule; for example, in the Sub case we finish by applying DeclSub. However, in the SpineRecover case we finish by applying DeclSpineRecover, but since DeclSpineRecover contains a premise that quantifies over all declarative derivations of a certain form, we must appeal to completeness! Consequently, soundness and completeness are really one theorem.

These parts are mutually recursive—later, we'll see that the DeclSpineRecover case of completeness must appeal to soundness (to show that the algorithmic type has no free existential variables). We cannot induct on the given derivation alone, because the derivations in the "for all" part of DeclSpineRecover are not subderivations. So we need a more involved induction measure that can make the leaps between soundness and completeness: lexicographic order with (1) the size of the subject term, (2) the judgment form, with ordinary spine judgments considered smaller than recovering spine judgments, and (3) the height of the derivation:

$$\left\langle e/s/\Pi, \quad \begin{array}{c}\text{ordinary spine judgment}\\ < \\ \text{recovering spine judgment}\end{array}, \quad \text{height}(\mathcal{D})\right\rangle$$

Proof sketch—SpineRecover case. By i.h., $[\Omega]\Gamma \vdash [\Omega]s : [\Omega]A\,! \gg [\Omega]C\; q$. Our goal is to apply DeclSpineRecover, which requires that we show that for *all* C' such that $[\Omega]\Theta \vdash s : [\Omega]A\,! \gg C'\; \not{!}$, we have $C' = [\Omega]C$. Suppose we have such a C'. By completeness (Theorem 12), $\Gamma \vdash s : [\Gamma]A\,! \gg C''\; q \dashv \Delta''$ where $\Delta'' \longrightarrow \Omega''$. We already have (as a subderivation) $\Gamma \vdash s : A\,! \gg C\; \not{!} \dashv \Delta$, so by determinacy, $C'' = C$ and $q = \not{!}$ and $\Delta'' = \Delta$. With the help of lemmas about context application, we can show $C' = [\Omega'']C'' = [\Omega'']C = [\Omega]C$. (Using completeness is permitted since our measure says a non-principality-restoring judgment is smaller.)

6.1 Auxiliary Soundness

For several auxiliary judgment forms, soundness is a matter of showing that, given two algorithmic terms, their declarative versions are equal. For example, for the instantiation judgment we have:

Lemma (Soundness of Instantiation).
If $\Gamma \vdash \hat{\alpha} := \tau : \kappa \dashv \Delta$ *and* $\hat{\alpha} \notin FV([\Gamma]\tau)$ *and* $[\Gamma]\tau = \tau$ *and* $\Delta \longrightarrow \Omega$ *then* $[\Omega]\hat{\alpha} = [\Omega]\tau$.

We have similar lemmas for term equality ($\Gamma \vdash \sigma \doteq t : \kappa \dashv \Delta$), propositional equivalence ($\Gamma \vdash P \equiv Q \dashv \Delta$) and type equivalence ($\Gamma \vdash A \equiv B \dashv \Delta$).

Our eliminating judgments incorporate assumptions into the context Γ. We show that the algorithmic rules for these judgments just append equations over universal variables:

Lemma (Soundness of Equality Elimination). *If* $[\Gamma]\sigma = \sigma$ *and* $[\Gamma]t = t$ *and* $\Gamma \vdash \sigma : \kappa$ *and* $\Gamma \vdash t : \kappa$ *and* $\mathsf{FEV}(\sigma) \cup \mathsf{FEV}(t) = \emptyset$, *then:*

(1) If $\Gamma\,/\,\sigma \doteq t : \kappa \dashv \Delta$ *then* $\Delta = (\Gamma, \Theta)$ *where* $\Theta = (\alpha_1 = t_1, \ldots, \alpha_n = t_n)$ *and for all* Ω *such that* $\Gamma \longrightarrow \Omega$ *and all* t' *s.t.* $\Omega \vdash t' : \kappa'$ *we have* $[\Omega, \Theta]t' = [\theta][\Omega]t'$ *where* $\theta = \mathrm{mgu}(\sigma, t)$.
(2) If $\Gamma\,/\,\sigma \doteq t : \kappa \dashv \bot$ *then no most general unifier exists.*

The last lemmas for soundness move directly from an algorithmic judgment to the corresponding declarative judgment.

Lemma (Soundness of Checkprop). *If* $\Gamma \vdash P$ *true* $\dashv \Delta$ *and* $\Delta \longrightarrow \Omega$ *then* $\Psi \vdash [\Omega]P$ *true.*

Lemma (Soundness of Match Coverage).

(1) If $\Gamma \vdash \Pi$ *covers* $\vec{A}\; q$ *and* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash \vec{A}\,!$ *types and* $[\Gamma]\vec{A} = \vec{A}$ *then* $[\Omega]\Gamma \vdash \Pi$ *covers* $\vec{A}\; q$.
(2) If $\Gamma\;/\;P \vdash \Pi$ *covers* $\vec{A}\,!$ *and* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash \vec{A}\,!$ *types and* $[\Gamma]\vec{A} = \vec{A}$ *and* $[\Gamma]P = P$ *then* $[\Omega]\Gamma\;/\;P \vdash \Pi$ *covers* $\vec{A}\,!$.

Theorem 6.9 (Soundness of Algorithmic Subtyping). *If* $[\Gamma]A = A$ *and* $[\Gamma]B = B$ *and* $\Gamma \vdash A$ *type and* $\Gamma \vdash B$ *type and* $\Delta \longrightarrow \Omega$ *and* $\Gamma \vdash A <:^{\mathcal{P}} B \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]A \leq^{\mathcal{P}} [\Omega]B$.

7 COMPLETENESS

We show that the algorithmic system is complete with respect to the declarative system. As with soundness, we need to show completeness of the auxiliary algorithmic judgments. We omit the full statements of these lemmas; as an example, if $[\Omega]\hat{\alpha} = [\Omega]\tau$ and $\hat{\alpha} \notin FV(\tau)$ then $\Gamma \vdash \hat{\alpha} := \tau : \kappa \dashv \Delta$.

7.1 Separation

To show completeness, we will need to show that wherever the declarative rule DeclSpineRecover is applied, we can apply the algorithmic rule SpineRecover. Thus, we need to show that *semantic* principality—that no other type can be given—entails that a type has no free existential variables.

The principality-recovering rules are potentially applicable when we start with a principal type $A\,!$ but produce $C\,\not{!}$, with Decl∀Spine changing $!$ to $\not{!}$. Completeness (Thm. 12) will use the "for all" part of DeclSpineRecover, which quantifies over all types produced by the spine rules under a given declarative context $[\Omega]\Gamma$. By i.h. we get an algorithmic spine judgment $\Gamma \vdash s : A'\,! \gg C'\,\not{!} \dashv \Delta$. Since A' is principal, unsolved existentials in C' must have been introduced within this derivation—they can't be in Γ already. Thus, we might have $\hat{\alpha} : \star \vdash s : A'\,! \gg \hat{\beta}\,\not{!} \dashv \hat{\alpha} : \star, \hat{\beta} : \star$ where a Decl∀Spine subderivation introduced $\hat{\beta}$, but $\hat{\alpha}$ can't appear in C'. We also can't equate $\hat{\alpha}$ and $\hat{\beta}$ in Δ, which would be tantamount to $C' = \hat{\alpha}$. Knowing that unsolved existentials in C' are "new" and independent from those in Γ means we can argue that, if there *were* an unsolved existential in C', it would correspond to an unforced choice in a Decl∀Spine subderivation, invalidating the "for all" part of DeclSpineRecover. Formalizing "must have been introduced" requires several definitions.

Definition 7.1 (Separation). An algorithmic context Γ is *separable into* $\Gamma_L * \Gamma_R$ if (1) $\Gamma = (\Gamma_L, \Gamma_R)$ and (2) for all $(\hat{\alpha} : \kappa = \tau) \in \Gamma_R$ it is the case that $\mathsf{FEV}(\tau) \subseteq \mathsf{dom}(\Gamma_R)$.

If Γ is separable into $\Gamma_L * \Gamma_R$, then Γ_R is self-contained in the sense that all existential variables declared in Γ_R have solutions whose existential variables are themselves declared in Γ_R. Every context Γ is separable into $\cdot * \Gamma$ and into $\Gamma * \cdot$.

Definition 7.2 (Separation-Preserving Ext.). Separated context $\Gamma_L * \Gamma_R$ extends to $\Delta_L * \Gamma_R$, written $(\Gamma_L * \Gamma_R) \xrightarrow[*]{} (\Delta_L * \Delta_R)$, if $(\Gamma_L, \Gamma_R) \longrightarrow (\Delta_L, \Delta_R)$ and $\mathsf{dom}(\Gamma_L) \subseteq \mathsf{dom}(\Delta_L)$ and $\mathsf{dom}(\Gamma_R) \subseteq \mathsf{dom}(\Delta_R)$.

Separation-preserving extension says that variables from one side of $*$ haven't "jumped" to the other side. Thus, Δ_L may add existential variables to Γ_L, and Δ_R may add existential variables to Γ_R, but no variable from Γ_L ends up in Δ_R and no variable from Γ_R ends up in Δ_L. It is necessary to write $(\Gamma_L * \Gamma_R) \xrightarrow[*]{} (\Delta_L * \Delta_R)$ rather than $(\Gamma_L * \Gamma_R) \longrightarrow (\Delta_L * \Delta_R)$, because only $\xrightarrow[*]{}$ includes the domain conditions. For example, $(\hat{\alpha} * \hat{\beta}) \longrightarrow (\hat{\alpha}, \hat{\beta} = \hat{\alpha}) * \cdot$, but $\hat{\beta}$ has jumped to the left of $*$ in the context $(\hat{\alpha}, \hat{\beta} = \hat{\alpha}) * \cdot$.

We prove many lemmas about separation, but use only one of them in the subsequent development (in the DeclSpineRecover case of typing completeness), and then only the part for spines. It says that if we have a spine whose type A mentions only variables in Γ_R, then the output context Δ extends Γ and preserves separation, and the output type C mentions only variables in Δ_R:

Lemma (Separation—Main). *If* $\Gamma_L * \Gamma_R \vdash s : A\,p \gg C\,q \dashv \Delta$ *or* $\Gamma_L * \Gamma_R \vdash s : A\,p \gg C\,\lceil q \rceil \dashv \Delta$ *and* $\Gamma_L * \Gamma_R \vdash A\,p$ *type and* $\mathsf{FEV}(A) \subseteq \mathsf{dom}(\Gamma_R)$ *then* $\Delta = (\Delta_L * \Delta_R)$ *and* $(\Gamma_L * \Gamma_R) \xrightarrow[*]{} (\Delta_L * \Delta_R)$ *and* $\mathsf{FEV}(C) \subseteq \mathsf{dom}(\Delta_R)$.

7.2 Completeness of Typing

Like soundness, completeness has several mutually recursive parts (see the appendix, p. 36).

Theorem 7.11 (Completeness of Algorithmic Typing). *Given* $\Gamma \longrightarrow \Omega$ *s.t.* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Omega)$*:*

(i) If $\Gamma \vdash A\,p$ *type and* $[\Omega]\Gamma \vdash [\Omega]e \Leftarrow [\Omega]A\,p$ *and* $p' \sqsubseteq p$ *then there exist* Δ *and* Ω' *such that* $\Delta \longrightarrow \Omega'$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash e \Leftarrow [\Gamma]A\,p' \dashv \Delta$.

(ii) If $\Gamma \vdash A\,p$ *type and* $[\Omega]\Gamma \vdash [\Omega]e \Rightarrow A\,p$ *then there exist* Δ, Ω', A', *and* $p' \sqsubseteq p$ *such that* $\Delta \longrightarrow \Omega'$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash e \Rightarrow A'\,p' \dashv \Delta$ *and* $A' = [\Delta]A'$ *and* $A = [\Omega']A'$.

(iii) If $\Gamma \vdash A\,p$ *type and* $[\Omega]\Gamma \vdash [\Omega]s : [\Omega]A\,p \gg B\,q$ *and* $p' \sqsubseteq p$ *then there exist* Δ, Ω', B', *and* $q' \sqsubseteq q$ *such that* $\Delta \longrightarrow \Omega'$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash s : [\Gamma]A\,p' \gg B'\,q' \dashv \Delta$ *and* $B' = [\Delta]B'$ *and* $B = [\Omega']B'$.

(iv) As part (iii), but with $\gg B\,\lceil q \rceil \cdots$ *and* $\gg B'\,\lceil q' \rceil \cdots$.

Proof sketch—DeclSpineRecover case. By i.h., $\Gamma \vdash s : [\Gamma]A\ ! \gg C'\ \not{\!\!\,\,} \dashv \Delta$ where $\Delta \longrightarrow \Omega'$ and $\Omega \longrightarrow \Omega'$ and $\mathrm{dom}(\Delta) = \mathrm{dom}(\Omega')$ and $C = [\Omega']C'$.

To apply SpineRecover, we need to show $\mathsf{FEV}([\Delta]C') = \emptyset$. Suppose, for a contradiction, that $\mathsf{FEV}([\Delta]C') \neq \emptyset$. Construct a variant of Ω' called Ω_2 that has a different solution for some $\hat{\alpha} \in \mathsf{FEV}([\Delta]C')$. By soundness (Thm. 12), $[\Omega_2]\Gamma \vdash [\Omega_2]s : [\Omega_2]A\ ! \gg [\Omega_2]C'\ \not{\!\!\,\,}$. Using a separation lemma with the trivial $\Gamma = (\Gamma * \cdot)$ we get $\Delta = (\Delta_L * \Delta_R)$ and $(\Gamma * \cdot) \xrightarrow[*]{} (\Delta_L * \Delta_R)$ and $\mathsf{FEV}(C') \subseteq \mathrm{dom}(\Delta_R)$. That is, all existentials in C' were introduced within the derivation of the (algorithmic) spine judgment. Thus, applying Ω_2 to things gives the same result as Ω, except for C', giving $[\Omega]\Gamma \vdash [\Omega]s : [\Omega]A\ ! \gg [\Omega_2]C'\ \not{\!\!\,\,}$. Now instantiate the "for all C_2" premise with $C_2 = [\Omega_2]C'$, giving $C = [\Omega_2]C'$. But we chose Ω_2 to have a different solution for $\hat{\alpha} \in \mathsf{FEV}(C')$, so we have $C \neq [\Omega_2]C'$: Contradiction. Therefore $\mathsf{FEV}([\Delta]C') = \emptyset$, so we can apply SpineRecover.

8 DISCUSSION AND RELATED WORK

A staggering amount of work has been done on GADTs and indexed types, and for space reasons we cannot offer a comprehensive survey of the literature. So we compare more deeply to fewer papers, to communicate our understanding of the design space.

Proof theory and type theory. As described in Section 1, there are two logical accounts of equality—the identity type of Martin-Löf and the equality type of Schroeder-Heister [1994] and Girard [1992]. The Girard/Schroeder-Heister equality has a more direct connection to pattern matching, which is why we make use of it. Coquand [1996] pioneered the study of pattern matching in dependent type theory. One perhaps surprising feature of Coquand's pattern-matching syntax is that it is strictly stronger than Martin-Löf's eliminators. His rules can derive the uniqueness of identity proofs as well as the disjointness of constructors. Constructor disjointness is also derivable from the Girard/Schroeder-Heister equality, because there is no unifier for two distinct constructors.

In future work, we hope to study the relation between these two notions of equality in more depth; richer equational theories (such as the theory of commutative rings or the $\beta\eta$-theory of the lambda calculus) do not have decidable unification, but it seems plausible that there are hybrid approaches which might let us retain some of the convenience of the G/SH equality rule while retaining the decidability of Martin-Löf's J eliminator.

Indexed and refinement types. Dependent ML [Xi and Pfenning 1999] indexed programs with propositional constraints, extending the ML type discipline to maintain additional invariants. DML collected constraints from the program and passed them to a constraint solver, a technique used by systems like Stardust [Dunfield 2007a] and liquid types [Rondon et al. 2008].

From phantom types to GADTs. Leijen and Meijer [1999] introduced the term *phantom type* to describe a technique for programming in ML/Haskell where additional type parameters are used to constrain when values are well-typed. This idea proved to have many applications, ranging from foreign function interfaces [Blume 2001] to encoding Java-style subtyping [Fluet and Pucella 2006]. Phantom types allow *constructing* values with constrained types, but do not easily permit *learning* about type equalities by *analyzing* them, putting applications such as intensional type analysis [Harper and Morrisett 1995] out of reach. Both Cheney and Hinze [2003] and Xi et al. [2003] proposed treating equalities as a first-class concept, giving explicitly-typed calculi for equalities, but without studying algorithms for type inference.

Simonet and Pottier [2007] gave a constraint-based algorithm for type inference for GADTs. It is this work which first identified the potential intractibility of type inference arising from the interaction of hypothetical constraints and unification variables. To resolve this issue they introduce the notion of *tractable* constraints (i.e., constraints where hypothetical equations never contain existentials), and require placing enough annotations that all constraints are tractable. In general, this could require annotations on case expressions, so subsequent work focused on relaxing this requirement. Though quite different in technical detail, stratified inference [Pottier and Régis-Gianas 2006] and *wobbly types* [Peyton Jones et al. 2006] both work by pushing type information from annotations to case expressions, with stratified type inference literally moving annotations around, and wobbly types tracking which parts of a type have no unification variables. Modern GHC uses the OutsideIn algorithm [Vytiniotis et al. 2011], which further relaxes the constraint: case analysis is permitted as long as it cannot modify what is known about an equation.

In our type system, the checking judgment of the bidirectional algorithm serves to propagate annotations; our requirement that the scrutinee of a case expression be principal ensures that no equations contain unification variables. The result is close in effect to stratified types, and is less expressive than OutsideIn. This is a deliberate design choice to keep the meaning of principality—that only a single type can be inferred for a term—clear and easy to understand.

To specify the OutsideIn approach, the case rule in our declarative system should permit scrutinizing an expression if all types that can be synthesized for it have exactly the same equations, even if they differ in their monotype parts. To achieve this, we would need to introduce a relation $C' \sim C$ which checks whether the equational constraints in C and C' are the same, and then modify the higher-order premise of the DeclSpineRecover rule to check that $C' \sim C$ (rather than $C' = C$, as it is currently). However, we thought such a spec is harder for programmers to develop an intuition for than simply saying that a scrutinee must synthesize a unique type.

Garrigue and Rémy [2013] proposed *ambivalent types*, which are a way of deciding when it is safe to generalize the type of a function using GADTs. This idea is orthogonal to our calculus, simply because we do no generalization at all: *every* polymorphic function takes an annotation. However, Garrigue and Rémy [2013] also emphasize the importance of *monotonicity*, which says that substitution should be stable under subtyping, that is, giving a more general type should not cause subtyping to fail. This condition is satisfied by our bidirectional system.

Karachalias et al. [2015] developed a coverage algorithm for GADTs that depends on external constraint solving; we offer a more self-contained but still logically-motivated approach.

Polarized subtyping. Barendregt et al. [1983] observed that a program which typechecks under a subtyping discipline can be checked without subtyping, provided that the program is sufficiently η-expanded. This idea of subtyping as η-expansion was investigated in a focused (albeit infinitary) setting by Zeilberger [2009]. Another notion of polarity arises from considering the (co-, contra-, in-)variance of type constructors. It is used by Abel [2006] to give a version of F^{ω} with subtyping, and Dolan and Mycroft [2017] apply this version of polarity to give a complete type inference algorithm for an ML-style language with subtyping. Our polarized subtyping judgment is closest in spirit to the work of Zeilberger [2009]. The restriction on our subtyping relation can be understood in terms of requiring the η expansions our subtyping relation infers to be in a focused normal form.

Extensions. To keep our formalization manageable, we left out some features that would be desirable in practice. In particular, we need (1) type constructors which take arguments and (2) recursive types [Pierce 2002, chapter 20]. The issue with both of these features is that they need to permit instantiating quantifiers with existentials and other binders, and our system relies upon monotypes (which do not contain such connectives). This limitation should create no difficulties in typical practice if we treat user-defined type constructors like List as monotypes, expanding the definition only as needed: when checking an expression against a user type constructor, and for pattern matching. Another extension, which we intend as future work, is to replace ordinary unification with pattern or nominal unification, to allow type instantiations containing binders.

Another extension is to increase the amount of type inference done. For instance, a natural question is whether we can extend the bidirectional approach to subsume the inference done by the algorithm of Damas and Milner [1982]. On the implementation side, this seems easy—to support ML-style type inference, we can add rules to infer types for values:

$$\frac{\Gamma, \blacktriangleright_{\hat{\alpha}}, \hat{\alpha}, \hat{\beta}, x : \hat{\alpha} \vdash e \Leftarrow \hat{\beta}\ \not! \dashv \Delta, \blacktriangleright_{\hat{\alpha}}, \Delta' \qquad \vec{\hat{\gamma}} = \mathsf{unsolved}(\Delta')}{\Gamma \vdash \lambda x.\, e \Rightarrow \forall \vec{\alpha}.\ [\vec{\alpha}/\vec{\hat{\gamma}}][\Delta'](\hat{\alpha} \to \hat{\beta})\ \mathbf{!} \dashv \Delta}$$

This rule adds a marker $\blacktriangleright_{\hat{\alpha}}$ to the context, then checks the body e against the type $\hat{\beta}$. Our output type substitutes away all the solved existential variables to the right of $\blacktriangleright_{\hat{\alpha}}$, and generalizes over all unsolved variables to the right of the marker. Using an ordered context gives precise control over the scope of the existential variables, easily expressing polymorphic generalization.

However, in the presence of generalization, the declarative specification of type inference no longer strictly specifies the order of polymorphic quantifiers (i.e., $\forall \alpha, \beta.\ \alpha \to \beta \to (\alpha \times \beta)$ and $\forall \beta, \alpha. \alpha \to \beta \to (\alpha \times \beta)$ should be equivalent) and so our principal synthesis would no longer return types stable up to alpha-equivalence. Fixing this would be straightforward (by relaxing the definition of type equivalence), but we have not pursued this because we do not value let-generalization enough to pay the price of increased complexity in our proofs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of this version, and of several previous versions, for their comments. We also thank Soham Chowdhury for his work on implementing the system presented in this paper. This work was partially funded by EPSRC grant EP/N02706X/2.

REFERENCES

Andreas Abel. 2006. Towards Generic Programming with Sized Types. In *Mathematics of Program Construction (LNCS)*, Tarmo Uustalu (Ed.), Vol. 4014. Springer, 10–28.

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2008. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction (MPC'08) (LNCS)*, Vol. 5133. Springer, 29–56.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symbolic Logic* 48, 4 (1983), 931–940.

Matthias Blume. 2001. No-Longer-Foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science* 59, 1 (2001).

James Cheney and Ralf Hinze. 2003. *First-Class Phantom Types.* Technical Report CUCIS TR2003-1901. Cornell University.

Jacek Chrząszcz. 1998. Polymorphic Subtyping Without Distributivity. In *Mathematical Foundations of Computer Science (LNCS)*, Vol. 1450. Springer, 346–355.

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Science of Computer Programming* 26, 1–3 (1996), 167–177.

Luis Damas and Robin Milner. 1982. Principal type-schemes for functional programs. In *POPL*. ACM Press, 207–212.

Rowan Davies and Frank Pfenning. 2000. Intersection Types and Computational Effects. In *ICFP*. ACM Press, 198–208.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *POPL*. ACM Press, 60–72.

Jana Dunfield. 2007a. Refined typechecking with Stardust. In *Programming Languages meets Programming Verification (PLPV '07)*. ACM Press, 21–32.

Jana Dunfield. 2007b. *A Unified System of Type Refinements.* Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-07-129.

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. In *ICFP*. ACM Press, 429–442. arXiv:1306.6032 [cs.PL].

Jana Dunfield and Frank Pfenning. 2003. Type Assignment for Intersections and Unions in Call-by-Value Languages. In *FoSSaCS*. Springer, 250–266.

Matthew Fluet and Riccardo Pucella. 2006. Phantom types and subtyping. (2006). arXiv:cs/0403034 [cs.PL].

Jacques Garrigue and Jacques Le Normand. 2015. GADTs and Exhaustiveness: Looking for the Impossible. In *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015 (EPTCS)*. 23–35.

Jacques Garrigue and Didier Rémy. 2013. Ambivalent Types for Principal Type Inference with GADTs. In *APLAS*. Springer, 257–272.

Jean-Yves Girard. 1992. A Fixpoint Theorem in Linear Logic. (1992). Post to Linear Logic mailing list, http://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html.

Robert Harper and Greg Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *POPL*. ACM Press, 130–141.

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. GADTs Meet Their Match: pattern-matching warnings that account for GADTs, guards, and laziness. In *ICFP*. ACM Press, 424–436.

Neelakantan R. Krishnaswami. 2009. Focusing on Pattern Matching. In *POPL*. ACM Press, 366–378.

Konstantin Läufer and Martin Odersky. 1994. Polymorphic type inference and abstract data types. *ACM Trans. Prog. Lang. Sys.* 16, 5 (1994), 1411–1430.

Daan Leijen and Erik Meijer. 1999. Domain specific embedded compilers. In *USENIX Conf. Domain-Specific Languages (DSL '99)*. ACM Press, 109–122.

Martin Odersky, Matthias Zenger, and Christoph Zenger. 2001. Colored Local Type Inference. In *POPL*. ACM Press, 41–53.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *J. Functional Programming* 17, 1 (2007), 1–82.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP*. ACM Press, 50–61.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*. ACM Press, 371–382.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Prog. Lang. Sys.* 22 (2000), 1–44.

François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *POPL*. ACM Press, 232–244.

Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*. ACM Press, 159–169.

Peter Schroeder-Heister. 1994. Definitional reflection and the completion. In *Extensions of Logic Programming (LNCS)*. Springer, 333–347.

Vincent Simonet and François Pottier. 2007. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1 (2007), 1.

Jerzy Tiuryn and Paweł Urzyczyn. 1996. The Subtyping Problem for Second-Order Types is Undecidable. In *LICS*. IEEE Press.

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalised. In *Workshop on Types in Language Design and Impl. (TLDI '10)*. ACM Press, 39–50.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *J. Functional Programming* 21, 4–5 (2011), 333–412.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2004. A Concurrent Logical Framework: The Propositional Fragment. In *Types for Proofs and Programs*. Springer LNCS 3085, 355–377.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL*. ACM Press, 224–235.

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL*. ACM Press, 214–227.

Noam Zeilberger. 2009. Refinement Types and Computational Duality. In *Programming Languages meets Programming Verification (PLPV '09)*. ACM Press, 15–26.