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

Joshua Dunfield
University of British Columbia
Vancouver, Canada
joshdunf@cs.ubc.ca

Neelakantan R. Krishnaswami
University of Birmingham
Birmingham, England
n.krishnaswami@cs.bham.ac.uk

Abstract

Bidirectional typechecking, in which terms either synthesize a type or are checked against a known type, has become popular for its scalability, 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 close to languages such as Haskell and OCaml. An explicit rule deduces when a type is principal, leading to reliable substitution principles 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.

1. Introduction

Consider an indexed sum type with a numeric index indicating whether the left or the right branch is inhabited, written in Haskell-like notation as follows:

```
data Sum : Nat -> * where
  Left   : A -> Sum 0
  Right  : B -> Sum (succ n)
```

We can use this definition to write a projection function that always gives us an element of A when the index is 0:

$$\begin{array}{l}\texttt{left} : \texttt{Sum}\ 0 \to A \\ \texttt{left}\ (\texttt{Left}\ a) = a\end{array}$$

This definition omits the clause for the `Right` branch. The `Right` branch has index $\mathsf{succ}(n)$ for some n, and the type annotation tells us that `left`'s argument has an index of 0. Since there exists no natural number n such that $0 = \mathsf{succ}(n)$, the `Right` branch cannot occur. Therefore it is safe to omit this case from the pattern match.

This is an entirely reasonable explanation for programmers, but language designers and implementors will have more questions. First, how can we implement such a type system? Clearly we needed some equality reasoning to justify leaving off the `Right` case, which is not trivial in general. Second, designers of functional languages are accustomed to the benefits of the Curry-Howard correspondence, and expect to see a *logical* reading of type systems to accompany the operational reading. So what is the logical reading of GADTs?

Since we relied on equality information to eliminate the second clause, it seems reasonable to look to logical accounts of equality. However, one of the ironies of proof theory is that it is possible to formulate equality in (at least) two different ways. The better-known approach is the *identity type* of Martin-Löf; another approach is due to Schroeder-Heister (1994) and Girard (1992).

Both approaches introduce equality via reflexivity:

$$\frac{}{\Gamma \vdash \mathsf{refl} : (t = t)}$$

However, they differ in the elimination rule. The Martin-Löf identity type eliminates equalities with an equality coercion J. A simplified (non-dependent) version of this rule can be formulated as:

$$\frac{\Gamma \vdash e : (s = t) \qquad \Gamma \vdash t : A(s)}{\Gamma \vdash \mathsf{J}(e, t) : A(t)}$$

Note that this kind of rule does not immediately justify the above definition of `left`: since J is a coercion, to use it we need to match the `Right` case and then show it leads to a contradiction.[1]

The elimination rule of Girard and Schroeder-Heister has a rather different flavour. It was originally formulated without proof terms, in a sequent calculus style:

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

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 $s = t$ by giving a proof of the goal C under each substitution θ that makes the two terms s and t equal.

There are three important features of the Girard/Schroeder-Heister rule, two good and one bad. First, when there are no unifiers, there are no premises: if we assume an inconsistent equation, we can immediately conclude the goal. For example, consider elim-

[1] Indeed, the identity type by itself is not enough to write the `left` function: without universes, MLTT cannot show that a proof of $(0 = 1)$ leads to a contradiction (Smith 1988)!

inating the equality $0 = 1$:

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

Second, this rule is an invertible left rule in the sequent calculus, which is known to correspond to pattern matching (Krishnaswami 2009). These two features line up nicely with our definition of `head`, where the impossibility of the `Left` case was indicated by the *absence* of a pattern clause. So it looks like the equalities used by GADTs correspond to the Girard/Schroeder-Heister equality, not the Martin-Löf identity type.

Alas, the third feature of this rule prevents us from just giving a proof term assignment for first-order logic and calling it a day. It is formulated in terms of unification, and 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. So we need to understand how to integrate these two uses of unification, or at least how to keep them decently apart, in order to take this logical specification and implement type inference for it.

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

Contributions. The equivalence of GADTs to the combination of existential types and equality constraints has long been known (Xi et al. 2003). Our fundamental contribution is to reduce GADTs to standard logical ingredients, *while retaining the implementability of the type system*. We accomplish this by formulating a system of indexed types in a bidirectional style (combining type *synthesis* with *checking* against a known type), which is well-known to combine practical implementability with theoretical tidiness.

- Our language supports implicit higher-rank polymorphism 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.

  Our system goes 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, because the unrestricted combination of universal and existential quantification seems to require mixed-prefix unification (i.e., solving equations under alternating quantifiers). We use the proof-theoretic technique of *focusing* to give a novel *polarized subtyping* judgment, which lets us treat alternating quantifiers in a way that retains decidability while maintaining other 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. This means that we do not need to write explicit equality proofs in our syntax, which permits us to more closely model the way equalities are used in OCaml and Haskell.
- Our calculus includes nested pattern matching, which fits neatly in the bidirectional framework, and allows a formal specification of coverage checking with GADTs.
- Our declarative system tracks whether or not a derivation has a principal type. The 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, it is also extremely non-algorithmic.
- We formulate an algorithmic type system (Section 4) for our declarative calculus, and prove that typechecking is decidable, deterministic (4.5), and sound and complete (Sections 5–6) with respect to the declarative system.

Supplementary material. We submitted *two* supplemental files: one with rules that we omitted for space reasons, and another (very long) file with detailed proofs and omitted lemma statements.

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. We explain our design choices by continuing with the `Sum` type definition from the introduction. As is well-known (Cheney and Hinze 2003; Xi et al. 2003), this kind of declaration can be desugared into type expressions that use equality and existential types to express the return type constraints; the example in the introduction desugars into something like

$$\mathtt{Sum}\ n \triangleq \big(A \times (n = 0)\big) + \big(\exists m : \mathbb{N}.\ B \times (n = \mathsf{succ}(m))\big)$$

While simple, this encoding suffices to illustrate all of the key difficulties in typechecking for GADTs.

Universal, existentials, and type inference. All 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 $\forall \alpha.\ \alpha \to \alpha$, and we want to apply it to the argument 3, then we want to write $f\ 3$, and not $f\ [\mathsf{Nat}]\ 3$ (as would be the case in pure System F). Even with a single type argument, this is a rather noisy style, and programs using even moderate amounts of polymorphism would 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 $\forall \alpha.\ \alpha \to \alpha$ can be viewed as a subtype of its instantiations (like $\mathsf{Nat} \to \mathsf{Nat}$).

For the subtype relation induced by polymorphism, subtype entailment is decidable (under modest restrictions). Matters get more complicated when *existential* types are also included. As can be seen in the encoding of the `Left` constructor of the $\mathtt{Sum}\ n$ type, existentials are necessary to encode equality constraints in GADTs. But the naive combination of existential and universal types requires doing unification under a *mixed prefix* of alternating quantifiers (Miller 1992), which is undecidable. Thus, programming languages traditionally have stringently restricted the use of existential types. They 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 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 $\exists \alpha. \alpha \to \alpha$).

While this is a practical solution, it increases the distance between surface languages and their type-theoretic cores. Our goal is to give a *direct* type-theoretic account of the features of our surface languages, avoiding complex elaboration passes.

The key problem in mixed-prefix unification is that the order in which to instantiate quantifiers is unclear. When deciding $\Gamma \vdash \forall \alpha.\ A(\alpha) \leq \exists b.\ B(b)$, we have the choice to choose an instantiation for α or for b, so that we prove the subtype entailment $\Gamma \vdash A(t) \leq \exists b.\ B(b)$ or the subtype entailment $\Gamma \vdash \forall \alpha.\ A(\alpha) \leq B(t)$. An algorithm will introduce a unification variable for α and then for b, or the other way around—and this choice matters! In the first order, b may depend on α, but not vice versa; with the second order, the allowed dependencies are reversed. Accurate dependency tracking amounts to Skolemization, which means we have reduced the problem to higher-order unification.

We adopt an idea from 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 easy).

The price we pay is that fewer subtype entailments are derivable. But the lost subtype entailments are those that rely on "clever" quantifier reversals, and all such entailments can be mimicked by writing identity coercions. So we do not lose fundamental expressivity, but we do gain decidability.

Equality as a property. The constructors in the datatype declaration above contain no explicit equality proofs: we can construct a value `Left` a without giving an equality proof that the index is zero. This is the usual convention in Haskell and OCaml, but our encoding pairs a value together with a proof. As before, 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 accurately, we treat equations $t = t'$ as properties, using a property type constructor $A \wedge P$ to model elements of type A satisfying the property (equation) P. (We also introduce $P \supset A$ for its adjoint dual.) So our encoding is really:

$$\texttt{Sum}\ n \triangleq \big(A \wedge (n = 0)\big) + \big(\exists m : \mathbb{N}.\, B \wedge (n = \mathsf{succ}(m))\big)$$

Then $\mathsf{inj}_1\ a$ inhabits `Sum` n only if the property $n = 0$ is true. Handling equality constraints through intersection types means 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).

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 is a popular implementation choice (for systems ranging from dependent types (Coquand 1996; Abel et al. 2008) to OO languages like C# and Scala (Bierman et al. 2007; 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 pattern matching (such as the omission of the `Right` case from the `left` function in the introduction).

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 unique. This property is lost once

$$\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid () \mid \lambda x.\, e \mid e_1\ (e_2 \cdot\!\cdot\, s) \mid (e : A) \\
 & & & \mid \langle e_1, e_2 \rangle \mid \mathsf{inj}_1\ e \mid \mathsf{inj}_2\ e \mid \mathsf{case}(e, \Pi) \\
\text{Values} & v & ::= & x \mid () \mid \lambda x.\, e \mid (v : A) \\
 & & & \mid \langle v_1, v_2 \rangle \mid \mathsf{inj}_1\ v \mid \mathsf{inj}_2\ v \\
\text{Spines} & s & ::= & \cdot \mid e \cdot\!\cdot\, s \\
\text{Patterns} & \rho & ::= & x \mid \langle \rho_1, \rho_2 \rangle \mid \mathsf{inj}_1\ \rho \mid \mathsf{inj}_2\ \rho \\
\text{Branches} & \pi & ::= & \vec{\rho} \Rightarrow e \\
\text{Lists of branches} & \Pi & ::= & \cdot \mid \big(\pi \mathrel{|} \Pi\big)
\end{array}$$

Figure 1. Source syntax

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

Figure 2. Syntax of declarative types and contexts

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. Declarative Typing

3.1 Syntax

Expression language. Expressions (Figure 1) are variables x, the unit value $()$, functions $\lambda x.\, e$, applications to *spines* $e_1\ (e_2 \cdot\!\cdot\, s)$, annotations $(e : A)$, pairs $\langle e_1, e_2 \rangle$, injections into a sum type $\mathsf{inj}_k\ e$, and case expressions $\mathsf{case}(e, \Pi)$ where Π is a list of branches π, which can eliminate pairs and injections; see below.

Values v are standard (for a call-by-value semantics).

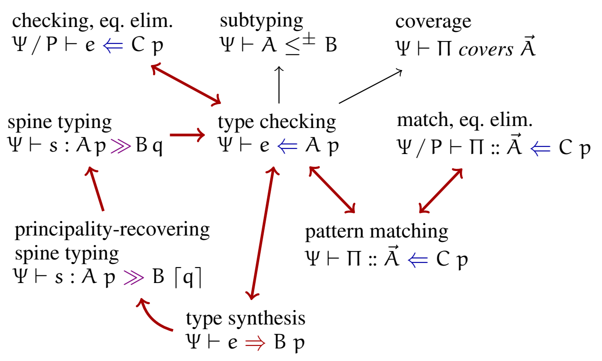
checking, eq. elim.
$\Psi / P \vdash e \Leftarrow C\ p$
subtyping
$\Psi \vdash A \leq^{\pm} B$
coverage
$\Psi \vdash \Pi\ covers\ \vec{A}$
spine typing
$\Psi \vdash s : A\,p \gg B\,q$
type checking
$\Psi \vdash e \Leftarrow A\ p$
match, eq. elim.
$\Psi / P \vdash \Pi :: \vec{A} \Leftarrow C\ p$
principality-recovering spine typing
$\Psi \vdash s : A\ p \gg B\ \lceil q \rceil$
pattern matching
$\Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p$
type synthesis
$\Psi \vdash e \Rightarrow B\ p$

Figure 3. Dependency structure of the declarative judgments

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

$$\frac{\Psi \vdash A\ \mathit{type} \qquad \mathit{nonpos}(A) \qquad \mathit{nonneg}(A)}{\Psi \vdash A \leq^{\pm} A}\ \leq\mathsf{Refl}\pm$$

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

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

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

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

Figure 4. Subtyping in the declarative system

A spine s is a list of expressions—arguments to a function. Allowing empty spines 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 $e_2 \cdot\cdot\, s$.

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 formulate pattern clauses as sequences as a technical convenience to specify pattern typing and coverage checking inductively, by letting us deconstruct tuple patterns into a sequence of 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* $A \wedge P$, read "A with P", which witnesses the proposition P. In both types, 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 $\mathsf{succ}(t)$ 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 3.3). 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.

3.2 Subtyping

We give our two subtyping relations 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 ($\leq\mathsf{Refl}$).

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. This also serves to illustrate a point of flexibility in the design of a bidirectional type system: we are relatively free to adjust the subtype relation to taste!

3.3 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 e is not the result of guessing. For example, the e in $(e : A)$ is checked against $A\,!$. In the synthesis judgment $\Psi \vdash e \Rightarrow A\ p$, 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 $e\ (e_1 \cdot\cdot\, \cdot)$. Supposing e_1 synthesizes $A_1 \to A_2$, we apply $\mathsf{Decl}{\to}\mathsf{Spine}$, checking e_1 against A_1 and using $\mathsf{DeclEmptySpine}$ to derive $\Psi \vdash \cdot : A_2\ p \gg A_2\ p$.

Rule $\mathsf{Decl}\forall\mathsf{Spine}$ does not decompose $e \cdot\cdot\, s$ but instantiates a $\forall$-quantifier. Note that, even if the given type $\forall\alpha : \kappa.\ A$ is principal ($p = !$), the type $[\tau/\alpha]A$ in the premise is not principal—we could choose a different τ. In fact, the q in $\mathsf{Decl}\forall\mathsf{Spine}$ 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 $\mathsf{Decl}\forall\mathsf{Spine}$ cannot affect the result type C. If A is principal ($p = !$) but the ordinary spine judgment produces a non-principal C, we can try to recover principality with $\mathsf{DeclSpineRecover}$. Its

$\boxed{\Psi \vdash e \Leftarrow A\ p}$ Under context Ψ, expression e checks against input type A

$\boxed{\Psi \vdash e \Rightarrow A\ p}$ Under context Ψ, expression e synthesizes output type A

$\boxed{\begin{array}{l}\Psi \vdash s : A\ p \gg C\ q \\ \Psi \vdash s : A\ p \gg C\ \lceil q \rceil\end{array}}$ 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

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

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

$$\frac{x : A\ p \in \Psi}{\Psi \vdash x \Rightarrow A\ p}\ \mathsf{DeclVar} \qquad \frac{\Psi \vdash e \Rightarrow A\ q \qquad \Psi \vdash A \leq^{\mathrm{pol}(B)} B}{\Psi \vdash e \Leftarrow B\ p}\ \mathsf{DeclSub} \qquad \frac{\Psi \vdash A\ \mathit{type} \qquad \Psi \vdash e \Leftarrow A\ !}{\Psi \vdash (e : A) \Rightarrow A\ !}\ \mathsf{DeclAnno}$$

$$\frac{}{\Psi \vdash () \Leftarrow 1\ p}\ \mathsf{Decl1I} \qquad \frac{v\ \mathit{chk\text{-}I} \qquad \Psi, \alpha : \kappa \vdash v \Leftarrow A\ p}{\Psi \vdash v \Leftarrow \forall \alpha : \kappa.\, A\ p}\ \mathsf{Decl\forall I} \qquad \frac{\Psi \vdash \tau : \kappa \qquad \Psi \vdash e \cdot\cdot s : [\tau/\alpha]A\ \not{!} \gg C\ q}{\Psi \vdash e \cdot\cdot s : \forall \alpha : \kappa.\, A\ p \gg C\ q}\ \mathsf{Decl\forall Spine}$$

$$\frac{\begin{array}{c}\Psi \vdash P\ \mathit{true} \\ \Psi \vdash e \Leftarrow A\ p\end{array}}{\Psi \vdash e \Leftarrow A \wedge P\ p}\ \mathsf{Decl\wedge I} \qquad \frac{v\ \mathit{chk\text{-}I} \qquad \Psi\ /\ P \vdash v \Leftarrow A\ !}{\Psi \vdash v \Leftarrow P \supset A\ !}\ \mathsf{Decl\supset I} \qquad \frac{\Psi \vdash P\ \mathit{true} \qquad \Psi \vdash e \cdot\cdot s : A\ p \gg C\ q}{\Psi \vdash e \cdot\cdot s : P \supset A\ p \gg C\ q}\ \mathsf{Decl\supset Spine}$$

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

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

$$\frac{}{\Psi \vdash \cdot : A\ p \gg A\ p}\ \mathsf{DeclEmptySpine} \qquad \frac{\begin{array}{c}\Psi \vdash e \Leftarrow A\ p \\ \Psi \vdash s : B\ p \gg C\ q\end{array}}{\Psi \vdash e \cdot\cdot s : A \rightarrow B\ p \gg C\ q}\ \mathsf{Decl{\rightarrow}Spine}$$

$$\frac{\Psi \vdash e \Leftarrow A_k\ p}{\Psi \vdash \mathsf{inj}_k\, e \Leftarrow A_1 + A_2\ p}\ \mathsf{Decl{+}I_k} \qquad \frac{\Psi \vdash e_1 \Leftarrow A_1\ p \qquad \Psi \vdash e_2 \Leftarrow A_2\ p}{\Psi \vdash \langle e_1, e_2 \rangle \Leftarrow A_1 \times A_2\ p}\ \mathsf{Decl{\times}I} \qquad \frac{\begin{array}{l}\Psi \vdash e \Rightarrow A\ ! \\ \Psi \vdash \Pi :: A \Leftarrow C\ p \\ \Psi \vdash \Pi\ \mathit{covers}\ A\end{array}}{\Psi \vdash \mathsf{case}(e, \Pi) \Leftarrow C\ p}\ \mathsf{DeclCase}$$

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

$$\frac{\mathsf{mgu}(\sigma, \tau) = \bot}{\Psi\ /\ (\sigma = \tau) \vdash e \Leftarrow C\ p}\ \mathsf{DeclCheck\bot} \qquad \frac{\mathsf{mgu}(\sigma, \tau) = \theta \qquad \theta(\Psi) \vdash \theta(e) \Leftarrow \theta(C)\ p}{\Psi\ /\ (\sigma = \tau) \vdash e \Leftarrow C\ p}\ \mathsf{DeclCheckUnify}$$

Figure 5. Declarative typing

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

$$\frac{}{\Psi \vdash \cdot :: \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatchEmpty} \qquad \frac{\Psi \vdash \pi :: \vec{A} \Leftarrow C\ p \qquad \Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p}{\Psi \vdash \pi \mid \Pi :: \Delta \Leftarrow C\ p}\ \mathsf{DeclMatchSeq} \qquad \frac{\Psi \vdash e \Leftarrow C\ p}{\Psi \vdash (\cdot \Rightarrow e) :: \cdot \Leftarrow C\ p}\ \mathsf{DeclMatchBase}$$

$$\frac{\Psi \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p}{\Psi \vdash (), \vec{\rho} \Rightarrow e :: 1, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatchUnit}$$

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

$$\frac{\Psi \vdash \rho, \vec{\rho} \Rightarrow e :: A_k, \vec{A} \Leftarrow C\ p}{\Psi \vdash \mathsf{inj}_k\, \rho, \vec{\rho} \Rightarrow e :: A_1 + A_2, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatch{+}_k} \qquad \frac{\Psi\ /\ P \vdash \vec{\rho} \Rightarrow e :: A, \vec{A} \Leftarrow C\ p}{\Psi \vdash \vec{\rho} \Rightarrow e :: A \wedge P, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatch\wedge}$$

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

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

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

Figure 6. Declarative pattern matching

$\boxed{\Psi \vdash \Pi\ \mathit{covers}\ \vec{A}}$ Patterns Π cover the types $\vec{A}$ in context Ψ

$$\frac{}{\Psi \vdash (\cdot \Rightarrow e_1) \mid \Pi'\ \mathit{covers}\ \cdot}\ \mathsf{DeclCoversEmpty} \qquad \frac{\Pi \overset{\mathrm{var}}{\rightsquigarrow} \Pi' \quad \Psi \vdash \Pi'\ \mathit{covers}\ \vec{A}}{\Psi \vdash \Pi\ \mathit{covers}\ A, \vec{A}}\ \mathsf{DeclCoversVar}$$

$$\frac{\Pi \overset{1}{\rightsquigarrow} \Pi' \quad \Psi \vdash \Pi'\ \mathit{covers}\ \vec{A}}{\Psi \vdash \Pi\ \mathit{covers}\ 1, \vec{A}}\ \mathsf{DeclCovers1} \qquad \frac{\Pi \overset{\times}{\rightsquigarrow} \Pi' \quad \Psi \vdash \Pi'\ \mathit{covers}\ A_1, A_2, \vec{A}}{\Psi \vdash \Pi\ \mathit{covers}\ A_1 \times A_2, \vec{A}}\ \mathsf{DeclCovers}\times$$

$$\frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R \quad \Psi \vdash \Pi_L\ \mathit{covers}\ A_1, \vec{A} \quad \Psi \vdash \Pi_R\ \mathit{covers}\ A_2, \vec{A}}{\Psi \vdash \Pi\ \mathit{covers}\ A_1 + A_2, \vec{A}}\ \mathsf{DeclCovers}+ \qquad \frac{\Psi, \alpha : \kappa \vdash \Pi\ \mathit{covers}\ \vec{A}}{\Psi \vdash \Pi\ \mathit{covers}\ \exists \alpha : \kappa.\ A, \vec{A}}\ \mathsf{DeclCovers}\exists$$

$$\frac{\theta = \mathsf{mgu}(t_1, t_2) \quad \theta(\Psi) \vdash \theta(\Pi)\ \mathit{covers}\ \theta(A_0, \vec{A})}{\Psi \vdash \Pi\ \mathit{covers}\ A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{DeclCoversEq} \qquad \frac{\mathsf{mgu}(t_1, t_2) = \bot}{\Psi \vdash \Pi\ \mathit{covers}\ A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{DeclCoversEqBot}$$

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

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

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

$$\frac{}{\cdot \overset{+}{\rightsquigarrow} \cdot \parallel \cdot} \qquad \frac{\rho \in \{u, \_\} \quad \Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{(\rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{+}{\rightsquigarrow} (\_, \vec{\rho} \Rightarrow e) \mid \Pi_L \parallel (\_, \vec{\rho} \Rightarrow e) \mid \Pi_R} \qquad \frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{(\mathsf{inj}_1\ \rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{+}{\rightsquigarrow} (\rho, \vec{\rho} \Rightarrow e) \mid \Pi_L \parallel \Pi_R}$$

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

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

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

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

Figure 7. Match coverage

first premise is $\Psi \vdash s : A\ \mathbf{!} \gg C\ \not{p}$; its second premise (really, an infinite set of premises) quantifies over all derivations of $\Psi \vdash s : A\ \mathbf{!} \gg C'\ \not{p}$. 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.

We need to stop and ask: Is DeclSpineRecover well-founded? If the second premise quantified over the same judgment form as the conclusion, certainly not; hence our distinction between the ordinary and recovery judgments. But the derivations in the second premise may have checking derivations (via Decl$\rightarrow$Spine), and in turn synthesis derivations, and in turn (via Decl$\rightarrow$E) the same recovery judgment! We are saved by the fact that Decl$\rightarrow$Spine and Decl$\rightarrow$E decompose their subject (the spine s or expression e)—any derivations of a recovery judgment lurking within the second premise of DeclSpineRecover must be for a smaller spine.

Pattern matching. The DeclCase rule checks that the scrutinee has a principal type, and then invokes the two main judgments for pattern matching. The $\Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p$ judgment checks that each branch in the list of branches Π is well-typed, and the $\Psi \vdash \Pi\ \mathit{covers}\ \vec{A}$ judgment does coverage checking for the list of clauses. Both of these judgments take a vector $\vec{A}$ of pattern types to simplify the specification of coverage checking.

The $\Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p$ judgment (in Figure 15) systematically checks the coverage of each clause in Π: the DeclMatchEmpty rule succeeds on the empty list, and the DeclMatchSeq rule checks one clause and recurs on the remaining elements.

The remaining rules for sums, units, and products break down patterns left to right, one constructor at a time. Products also extend the pattern and type sequences, with DeclMatch$\times$ breaking down a pattern vector headed by a pair pattern $\langle p, p' \rangle, \vec{p}$ into $p, p', \vec{p}$ (also turning the type sequence from $A \times B, \vec{C}$ into $A, B, \vec{C}$). Once all the patterns are eliminated, the DeclMatchBase rule says that if the body typechecks, then the clause typechecks. For completeness, the variable and wildcard rules are both 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 the branch always 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.

The $\Psi \vdash \Pi\ \mathit{covers}\ \vec{A}$ judgment (in Figure 16) checks whether a set of patterns covers all the possible cases. As with match typing, we systematically deconstruct the sequence of types in the pattern clause, but this time, we need a set of 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, $\mathsf{DeclCovers}\times$ 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 $\mathsf{DeclCovers}+$ can check the coverage of the left and right branches independently.

As with typing, $\mathsf{DeclCovers}\exists$ just unpacks the existential type, Likewise, $\mathsf{DeclCoversEqBot}$ and $\mathsf{DeclCoversEq}$ 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.

4. Algorithmic Typing

Our algorithmic system mimics our declarative rules as closely as possible, with one key difference: whenever the declarative system would make a guess, we introduce an existential variable into the context (written with a hat $\hat{\alpha}$). As typechecking proceeds, we refine the value of the existential variables to reflect our increasing knowledge. This means that each of the declarative typing judgments has a corresponding algorithmic judgment taking both an input and an output context: the type checking judgment $\Gamma \vdash e \Leftarrow A\ p \dashv \Delta$ now takes an input context Γ and yields an output context Δ reflecting our increased knowledge of what the types have to be. A judgment $\Gamma \longrightarrow \Delta$, explained in Section 4.4, formalizes the notion of increasing knowledge.

These judgments are documented in Figure 13, which has a dependency graph of the algorithmic judgments. Each declarative judgment has a corresponding algorithmic judgment, but the algorithmic system adds a few more judgments, such as type 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 the process of solving existential variables to equate terms.

We give the algorithmic typing rules in Figure 12, but rules for most other judgments are in the supplemental appendix.

4.1 Syntax

Expression language. The expression language is 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 8). We use the same meta-variables, e.g. A, B, C for types. 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:

- *unsolved* existential variable declarations $\hat{\alpha} : \kappa$ (included in the $\Gamma, u : \kappa$ production);
- *solved* existential variable declarations $\hat{\alpha} : \kappa = \tau$;
- *equations* over universal variables $\alpha = \tau$; and
- *markers* $\blacktriangleright_u$.

Universal variables	α, β, γ		
Existential variables	$\hat{\alpha}, \hat{\beta}, \hat{\gamma}$		
Variables	u	$::=$	$\alpha \mid \hat{\alpha}$
Types	A, B, C	$::=$	$1 \mid \alpha \mid \hat{\alpha}$
			$\mid \forall \alpha : \kappa.\, A \mid \exists \alpha : \kappa.\, A$
			$\mid P \supset A \mid A \wedge P$
			$\mid A \to B \mid A + B \mid A \times B$
Propositions	P, Q	$::=$	$t = t'$
Binary connectives	$\oplus$	$::=$	$\to \mid + \mid \times$
Terms/monotypes	t, τ, σ	$::=$	$\mathsf{zero} \mid \mathsf{succ}(t) \mid 1 \mid \alpha \mid \hat{\alpha}$
			$\mid \tau \to \sigma \mid \tau + \sigma \mid \tau \times \sigma$
Contexts	Γ, Δ, Θ	$::=$	$\cdot \mid \Gamma, u : \kappa \mid \Gamma, x : A\, p$
			$\mid \Gamma, \hat{\alpha} : \kappa = \tau \mid \Gamma, \alpha = t \mid \Gamma, \blacktriangleright_u$
Complete contexts	Ω	$::=$	$\cdot \mid \Omega, \alpha : \kappa \mid \Omega, x : A\, p$
			$\mid \Omega, \hat{\alpha} : \kappa = \tau \mid \Omega, \alpha = t \mid \Omega, \blacktriangleright_u$
Possibly-inconsistent contexts	$\Delta^\perp$	$::=$	$\Delta \mid \perp$

Figure 8. Syntax of types, contexts, and other objects in the algorithmic system

$$
\begin{array}{lcl}
[\Gamma]1 & = & 1 \\
[\Gamma]\alpha & = & \begin{cases} [\Gamma]\tau & \text{when } (\alpha = \tau) \in \Gamma \\ \alpha & \text{otherwise} \end{cases} \\
\big[\Gamma[\hat{\alpha} : \kappa = \tau]\big]\hat{\alpha} & = & [\Gamma]\tau \\
\big[\Gamma[\hat{\alpha} : \kappa]\big]\hat{\alpha} & = & \hat{\alpha} \\
[\Gamma](A \supset B) & = & ([\Gamma]A) \supset ([\Gamma]B) \\
[\Gamma](A \wedge B) & = & ([\Gamma]A) \wedge ([\Gamma]B) \\
[\Gamma](A \oplus B) & = & ([\Gamma]A) \oplus ([\Gamma]B) \\
[\Gamma](\forall \alpha : \kappa.\, A) & = & \forall \alpha : \kappa.\, [\Gamma]A \\
[\Gamma](\exists \alpha : \kappa.\, A) & = & \exists \alpha : \kappa.\, [\Gamma]A
\end{array}
$$

Figure 9. Applying a context, as a substitution, to a type

An equation $\alpha = \tau$ must appear to the right of the universal variable's declaration $\alpha : \kappa$.

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

Possibly-inconsistent contexts. 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 $\perp$.

4.2 Context substitution

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 to type A; this operation is defined in Figure 9.

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. This operation of context application $[\Omega]\Gamma$ is given in Figure 10.

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

Figure 10. Applying a complete context Ω to a context

The application $[\Omega]\Gamma$ is defined if and only if $\Gamma \longrightarrow \Omega$ (context extension; see Section 4.4), and applying Ω to any such Γ yields the same declarative context $[\Omega]\Omega$.

Complete contexts are essential for stating and proving soundness and completeness, but are not explicitly distinguished in any rules.

4.3 Hole notation

Since we will manipulate contexts not only by appending declarations (as in the declarative system) but by inserting and replacing declarations in the middle, 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 $\Gamma_0[\hat{\beta} = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta})$. Since this notation is concise, we use it even in rules that do not replace declarations, such as the rules for type well-formedness.

Occasionally, we also need contexts with *two* ordered holes:

$$\Gamma = \Gamma_0[\Theta_1][\Theta_2] \quad \text{means } \Gamma \text{ has the form } (\Gamma_L, \Theta_1, \Gamma_M, \Theta_2, \Gamma_R)$$

4.4 The context extension relation

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.

We can also interpret $\Gamma \longrightarrow \Delta$ as saying that Γ is *entailed by* Δ: all positive information derivable from Γ (say, that existential variable $\hat{\alpha}$ is in scope) can also be derived from Δ (which may have more information, say, that $\hat{\alpha}$ is equal to a particular type).

The rules deriving the context extension judgment (Figure 11) 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 must match ($\longrightarrow$Uvar); equations on universal variables must match ($\longrightarrow$Eqn); scope markers must match ($\longrightarrow$Marker); and, existential variables may:

- be unsolved in both contexts ($\longrightarrow$Unsolved),
- be solved in both contexts, if applying the extending context Δ makes the solutions equal ($\longrightarrow$Solved),
- get solved by the extending context ($\longrightarrow$Solve),
- be added by the extending context, either without a solution ($\longrightarrow$Add) or with a solution ($\longrightarrow$AddSolved);

Extension allows solutions to change, if information is preserved or increased. The extension

$$(\hat{\alpha} : \star, \hat{\beta} : \star = \hat{\alpha}) \longrightarrow (\hat{\alpha} : \star = 1, \hat{\beta} : \star = \hat{\alpha})$$

directly increases information about $\hat{\alpha}$, and indirectly increases information about $\hat{\beta}$. Perhaps more interestingly, the extension

$$\underbrace{(\hat{\alpha} : \star = 1, \hat{\beta} : \star = \hat{\alpha})}_{\Delta} \longrightarrow \underbrace{(\hat{\alpha} : \star = 1, \hat{\beta} : \star = 1)}_{\Omega}$$

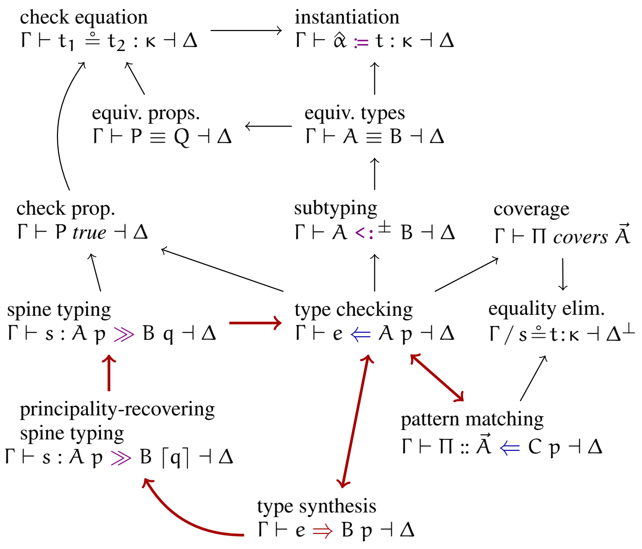
check equation
$\Gamma \vdash t_1 \stackrel{\circ}{=} t_2 : \kappa \dashv \Delta$
instantiation
$\Gamma \vdash \hat{\alpha} := t : \kappa \dashv \Delta$
equiv. props.
$\Gamma \vdash P \equiv Q \dashv \Delta$
equiv. types
$\Gamma \vdash A \equiv B \dashv \Delta$
check prop.
$\Gamma \vdash P\ \mathit{true} \dashv \Delta$
subtyping
$\Gamma \vdash A <:^{\pm} B \dashv \Delta$
coverage
$\Gamma \vdash \Pi\ \mathit{covers}\ \vec{A}$
spine typing
$\Gamma \vdash s : A\,p \gg B\,q \dashv \Delta$
type checking
$\Gamma \vdash e \Leftarrow A\,p \dashv \Delta$
equality elim.
$\Gamma / s \stackrel{\circ}{=} t : \kappa \dashv \Delta^{\perp}$
principality-recovering spine typing
$\Gamma \vdash s : A\,p \gg B\,\lceil q \rceil \dashv \Delta$
pattern matching
$\Gamma \vdash \Pi :: \vec{A} \Leftarrow C\,p \dashv \Delta$
type synthesis
$\Gamma \vdash e \Rightarrow B\,p \dashv \Delta$

Figure 13. Dependency structure of the algorithmic judgments

also holds: 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 two solutions gives the same thing: applying Ω to Δ's solution of $\hat{\beta}$ gives $[\Omega]\hat{\alpha} = [\Omega]1 = 1$, while applying Ω to Ω's own solution for $\hat{\beta}$ also gives 1, because $[\Omega]1 = 1$.

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.

This combination of rigidity (in demanding that the order of declarations be preserved) with flexibility (in how existential type variable solutions are expressed) manages to satisfy scoping and dependency relations *and* give enough room to manoeuvre in the algorithm and metatheory.

4.5 Determinacy

Our algorithmic judgments have the nice property that, given appropriate inputs, only one set of outputs is derivable. In addition to being nice, we use this property in the proof of soundness, for spine judgments:

Theorem 1 (Determinacy of Typing)**.** *Given* Γ, e, A, p *such that* $\Gamma \vdash e : A\,p \gg C_1\,q_1 \dashv \Delta_1$ *and* $\Gamma \vdash e : A\,p \gg C_2\,q_2 \dashv \Delta_2$, *it is the case that* $C_1 = C_2$ *and* $q_1 = q_2$ *and* $\Delta_1 = \Delta_2$.

5. Soundness

We show that the algorithmic system is sound with respect to the declarative system.

5.1 Equating lemmas

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:

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

$$\dfrac{}{\cdot \longrightarrow \cdot}\ \longrightarrow\!\mathsf{Id} \qquad \dfrac{\Gamma \longrightarrow \Delta \qquad [\Delta]A = [\Delta]A'}{\Gamma, x : A\,p \longrightarrow \Delta, x : A'\,p}\ \longrightarrow\!\mathsf{Var} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \alpha : \kappa \longrightarrow \Delta, \alpha : \kappa}\ \longrightarrow\!\mathsf{Uvar} \qquad \dfrac{\Gamma \longrightarrow \Delta \qquad [\Delta]t = [\Delta]t'}{\Gamma, \alpha = t \longrightarrow \Delta, \alpha = t'}\ \longrightarrow\!\mathsf{Eqn}$$

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

$$\dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat\alpha : \kappa}\ \longrightarrow\!\mathsf{Add} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \hat\alpha : \kappa = t}\ \longrightarrow\!\mathsf{AddSolved} \qquad \dfrac{\Gamma \longrightarrow \Delta}{\Gamma, \blacktriangleright_u \longrightarrow \Delta, \blacktriangleright_u}\ \longrightarrow\!\mathsf{Marker}$$

Figure 11. Context extension

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

$\boxed{\Gamma \vdash e \Rightarrow A\,p \dashv \Delta}$ Under input context Γ, expression e synthesizes output type A, with output context Δ

$\boxed{\begin{array}{l}\Gamma \vdash s : A\,p \gg C\,q \dashv \Delta \\ \Gamma \vdash s : A\,p \gg C\,\lceil q\rceil \dashv \Delta\end{array}}$ 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

$$\dfrac{(x : A\,p) \in \Gamma}{\Gamma \vdash x \Rightarrow [\Gamma]A\,p \dashv \Gamma}\ \mathsf{Var} \qquad \dfrac{\Gamma \vdash e \Rightarrow A\,q \dashv \Theta \qquad \Theta \vdash A <:^{\mathrm{pol}(B)} B \dashv \Delta}{\Gamma \vdash e \Leftarrow B\,p \dashv \Delta}\ \mathsf{Sub} \qquad \dfrac{\Gamma \vdash A\,!\ \mathit{type} \qquad \Gamma \vdash e \Leftarrow [\Gamma]A\,! \dashv \Delta}{\Gamma \vdash (e : A) \Rightarrow [\Delta]A\,! \dashv \Delta}\ \mathsf{Anno}$$

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

$$\dfrac{v\ \mathit{chk\text{-}I} \qquad \Gamma, \alpha : \kappa \vdash v \Leftarrow A\,p \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash v \Leftarrow \forall\alpha : \kappa.\, A\,p \dashv \Delta}\ \forall\mathsf{I} \qquad \dfrac{\Gamma, \hat\alpha : \kappa \vdash e \cdot\!\cdot\, s : [\hat\alpha/\alpha]A \gg C\,q \dashv \Delta}{\Gamma \vdash e \cdot\!\cdot\, s : \forall\alpha : \kappa.\, A\,p \gg C\,q \dashv \Delta}\ \forall\mathsf{Spine}$$

$$\dfrac{e \text{ not a case} \qquad \Gamma \vdash P\ \mathit{true} \dashv \Theta \qquad \Theta \vdash e \Leftarrow [\Theta]A\,p \dashv \Delta}{\Gamma \vdash e \Leftarrow A \wedge P\,p \dashv \Delta}\ \wedge\mathsf{I} \qquad \dfrac{v\ \mathit{chk\text{-}I} \qquad \Gamma, \blacktriangleright_P \,/\, P \dashv \Theta \qquad \Theta \vdash v \Leftarrow [\Theta]A\,! \dashv \Delta, \blacktriangleright_P, \Delta'}{\Gamma \vdash v \Leftarrow P \supset A\,! \dashv \Delta}\ \supset\!\mathsf{I}$$

$$\dfrac{v\ \mathit{chk\text{-}I} \qquad \Gamma, \blacktriangleright_P \,/\, P \dashv \bot}{\Gamma \vdash v \Leftarrow P \supset A\,! \dashv \Gamma}\ \supset\!\mathsf{I}\bot \qquad \dfrac{\Gamma \vdash P\ \mathit{true} \dashv \Theta \qquad \Theta \vdash e \cdot\!\cdot\, s : [\Theta]A\,p \gg C\,q \dashv \Delta}{\Gamma \vdash e \cdot\!\cdot\, s : P \supset A\,p \gg C\,q \dashv \Delta}\ \supset\!\mathsf{Spine}$$

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

$$\dfrac{\Gamma \vdash e \Rightarrow A\,p \dashv \Theta \qquad \Theta \vdash s : A\,p \gg C\,\lceil q\rceil \dashv \Delta}{\Gamma \vdash e\,s \Rightarrow C\,q \dashv \Delta}\ \to\!\mathsf{E}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash s : A\,! \gg C\,\cancel{!} \dashv \Delta \\ \mathsf{FEV}(C) = \emptyset\end{array}}{\Gamma \vdash s : A\,! \gg C\,\lceil !\rceil \dashv \Delta}\ \mathsf{SpineRecover} \qquad \dfrac{\begin{array}{l}\Gamma \vdash s : A\,p \gg C\,q \dashv \Delta \\ \big((p = \cancel{!}) \text{ or } (q = !) \text{ or } (\mathsf{FEV}(C) \neq \emptyset)\big)\end{array}}{\Gamma \vdash s : A\,p \gg C\,\lceil q\rceil \dashv \Delta}\ \mathsf{SpinePass}$$

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

$$\dfrac{\Gamma \vdash e \Leftarrow A_k\,p \dashv \Delta}{\Gamma \vdash \mathsf{inj}_k\, e \Leftarrow A_1 + A_2\,p \dashv \Delta}\ +\mathsf{I}_k \qquad \dfrac{\Gamma[\hat\alpha_1 : \star, \hat\alpha_2 : \star, \hat\alpha : \star = \hat\alpha_1{+}\hat\alpha_2] \vdash e \Leftarrow \hat\alpha_k \dashv \Delta}{\Gamma[\hat\alpha : \star] \vdash \mathsf{inj}_k\, e \Leftarrow \hat\alpha \dashv \Delta}\ +\mathsf{I}\hat\alpha_k$$

$$\dfrac{\Gamma \vdash e_1 \Leftarrow A_1\,p \dashv \Theta \qquad \Theta \vdash e_2 \Leftarrow [\Theta]A_2\,p \dashv \Delta}{\Gamma \vdash \langle e_1, e_2\rangle \Leftarrow A_1 \times A_2\,p \dashv \Delta}\ \times\mathsf{I} \qquad \dfrac{\begin{array}{l}\Gamma[\hat\alpha_2{:}\star, \hat\alpha_1{:}\star, \hat\alpha{:}\star = \hat\alpha_1{\times}\hat\alpha_2] \vdash e_1 \Leftarrow \hat\alpha_1 \dashv \Theta \\ \Theta \vdash e_2 \Leftarrow [\Theta]\hat\alpha_2 \dashv \Delta\end{array}}{\Gamma[\hat\alpha : \star] \vdash \langle e_1, e_2\rangle \Leftarrow \hat\alpha \dashv \Delta}\ \times\mathsf{I}\hat\alpha$$

$$\dfrac{\Gamma[\hat\alpha_2 : \star, \hat\alpha_1 : \star, \hat\alpha : \star = \hat\alpha_1{\to}\hat\alpha_2] \vdash e \cdot\!\cdot\, s : (\hat\alpha_1 \to \hat\alpha_2) \gg C \dashv \Delta}{\Gamma[\hat\alpha : \star] \vdash e \cdot\!\cdot\, s : \hat\alpha \gg C \dashv \Delta}\ \hat\alpha\mathsf{Spine}$$

$$\dfrac{\Gamma \vdash e \Rightarrow A\,! \dashv \Theta \qquad \Theta \vdash \Pi :: [\Theta]A \Leftarrow [\Theta]C\,p \dashv \Delta \qquad \Delta \vdash \Pi\ \mathit{covers}\ [\Delta]A}{\Gamma \vdash \mathsf{case}(e, \Pi) \Leftarrow C\,p \dashv \Delta}\ \mathsf{Case}$$

Figure 12. Algorithmic typing

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

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

5.2 Elimination lemmas

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 $\Gamma \;/\; \sigma \stackrel{\circ}{=} t : \kappa \dashv \Delta$
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* $\theta = \mathsf{mgu}(\sigma, t)$.
(2) If $\Gamma \;/\; \sigma \stackrel{\circ}{=} t : \kappa \dashv \bot$ *then no most general unifier exists.*

5.3 Direct lemmas

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 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* $\Gamma \vdash A <:^{\pm} B \dashv \Delta$ *then* $[\Omega]\Delta \vdash [\Omega]A \leq^{\pm} [\Omega]B$.

Lemma (Soundness of Match Coverage).
If $\Gamma \vdash \Pi$ *covers* $\vec{A}$ *and* $\Gamma \longrightarrow \Omega$ *and* $\Gamma \vdash \vec{A}$ **!** *types and* $[\Gamma]\vec{A} = \vec{A}$ *then* $[\Omega]\Gamma \vdash \Pi$ *covers* $\vec{A}$.

5.4 Soundness of typing

With lemmas for all the auxiliary judgments in hand, we can move on to the main soundness result. It has six mutually-recursive parts, one for each of the checking, synthesis, spine, and match judgments—including the principality-recovering spine judgment and the assumption-adding match judgment.

Theorem 2 (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 $\Gamma \vdash \Pi :: \vec{A} \Leftarrow C\ p \dashv \Delta$ *and* $\Gamma \vdash \vec{A}$ **!** *types and* $[\Gamma]\vec{A} = \vec{A}$ *and* $\Gamma \vdash C\ p$ *type*
then $[\Omega]\Delta \vdash [\Omega]\Pi :: [\Omega]\vec{A} \Leftarrow [\Omega]C\ p$.
(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 is simply "turning the crank": applying the induction hypothesis to each premise, yielding derivations of corresponding declarative judgments (with Ω applied to everything in sight), and applying the corresponding declarative rule; for example, in the Sub case we finish the proof by applying DeclSub. The SpineRecover case is interesting: we do 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 two parts of 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:

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

Proof sketch*—SpineRecover *case. By i.h., $[\Omega]\Gamma \vdash [\Omega]s : [\Omega]A\ \mathbf{!} \gg [\Omega]C\ q$. Our goal is to apply DeclSpineRecover, which requires that we show that for *all* C' such that $[\Omega]\Theta \vdash s : [\Omega]A\ \mathbf{!} \gg C'\ \not{!}$, we have $C' = [\Omega]C$. Suppose we have such a C'. By completeness (Theorem 3), $\Gamma \vdash s : [\Gamma]A\ \mathbf{!} \gg C''\ q \dashv \Delta''$ where $\Delta'' \longrightarrow \Omega''$. We already have (as a subderivation) $\Gamma \vdash s : A\ \mathbf{!} \gg C\ \not{!} \dashv \Delta$, 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—really, using the i.h.—is justified because our measure considers a non-principality-restoring judgment to be smaller.

6. 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, but as an example, if $[\Omega]\hat{\alpha} = [\Omega]\tau$ then $\Gamma \vdash \hat{\alpha} := \tau : \kappa \dashv \Delta$, under certain conditions (including that $\hat{\alpha} \notin \mathrm{FV}(\tau)$).

6.1 Separation

To show completeness, we will need to show that wherever the declarative rule DeclSpineRecover is applied, we can apply the algorithmic rule SpineRecover. More concretely, the *semantic* notion of principality—that no other type can be given—must entail the *syntactic* notion that a type has no free existential variables.

The principality-recovering rules are potentially applicable when we start with a principal type $A\ \mathbf{!}$ but produce $C\ \not{!}$, with Decl∀Spine changing **!** to $\not{!}$. The proof of completeness (Thm. 3) 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 $\Gamma \vdash s : A'\ \mathbf{!} \gg C'\ \not{!} \dashv \Delta$. Since A' is principal, any unsolved existentials in C' must have been introduced within this derivation—they can't be in Γ already. Thus, we might have $\hat{\alpha} : \star \vdash s : A'\ \mathbf{!} \gg \hat{\beta}\ \not{!} \dashv \hat{\alpha} : \star, \hat{\beta} : \star$ 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 morally equivalent 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 claims like "must have been introduced" requires several definitions.

Definition 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 2 (Separation-Preserving Extension).
The separated context $\Gamma_L * \Gamma_R$ *extends to* $\Delta_L * \Delta_R$*, written*

$$(\Gamma_L * \Gamma_R) \underset{*}{\longrightarrow} (\Delta_L * \Delta_R)$$

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 $(\Gamma_L * \Gamma_R) \underset{*}{\longrightarrow} (\Delta_L * \Delta_R)$ rather than $(\Gamma_L * \Gamma_R) \longrightarrow (\Delta_L * \Delta_R)$, because only $\underset{*}{\longrightarrow}$ 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* $(\Gamma_L * \Gamma_R) \underset{*}{\longrightarrow} (\Delta_L * \Delta_R)$ *and* $\mathsf{FEV}(C) \subseteq \mathsf{dom}(\Delta_R)$.

6.2 Completeness of typing

Theorem 3 (Completeness of Algorithmic Typing).
Given $\Gamma \longrightarrow \Omega$ *such that* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Omega)$*:*

(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) If $\Gamma \vdash A\ p$ *type and* $[\Omega]\Gamma \vdash [\Omega]s : [\Omega]A\ p \gg B\ \lceil q \rceil$ *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* $\Gamma \vdash s : [\Gamma]A\ p' \gg B'\ \lceil q' \rceil \dashv \Delta$ *and* $B' = [\Delta]B'$ *and* $B = [\Omega']B'$.

(v) If $\Gamma \vdash \vec{A}\ !$ *types and* $\Gamma \vdash C\ p$ *type and* $[\Omega]\Gamma \vdash [\Omega]\Pi :: [\Omega]\vec{A} \Leftarrow [\Omega]C\ p$ *and* $p' \sqsubseteq p$ *then there exist* Δ, Ω', *and* C *such that* $\Delta \longrightarrow \Omega'$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma \vdash \Pi :: [\Gamma]\vec{A} \Leftarrow [\Gamma]C\ p' \dashv \Delta$.

(vi) If $\Gamma \vdash \vec{A}\ !$ *types and* $\Gamma \vdash P$ *prop and* $\mathsf{FEV}(P) = \emptyset$ *and* $\Gamma \vdash C\ p$ *type and* $[\Omega]\Gamma\ /\ [\Omega]P \vdash [\Omega]\Pi :: [\Omega]\vec{A} \Leftarrow [\Omega]C\ p$ *and* $p' \sqsubseteq p$ *then there exist* Δ, Ω', *and* C *such that* $\Delta \longrightarrow \Omega'$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ *and* $\Omega \longrightarrow \Omega'$ *and* $\Gamma\ /\ [\Gamma]P \vdash \Pi :: [\Gamma]\vec{A} \Leftarrow [\Gamma]C\ p' \dashv \Delta$.

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 $\mathsf{dom}(\Delta) = \mathsf{dom}(\Omega')$ 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. 3), $[\Omega_2]\Gamma \vdash [\Omega_2]s : [\Omega_2]A\ ! \gg [\Omega_2]C'\ \not{!}$. Using the separation lemma with the trivial separation $\Gamma = (\Gamma * \cdot)$ we get $\Delta = (\Delta_L * \Delta_R)$ and $(\Gamma * \cdot) \underset{*}{\longrightarrow} (\Delta_L * \Delta_R)$ and $\mathsf{FEV}(C') \subseteq \mathsf{dom}(\Delta_R)$. 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.

7. 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 Axiom K (uniqueness of identity proofs) as well as the disjointness of constructors. Similarly, constructor disjointness is derivable from the Girard/Schroeder-Heister equality, because unification fails when two distinct constructors are compared.

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, catching bugs in programs that type-check under the standard ML type discipline but fail to maintain additional invariants tracked by the propositional annotations. DML worked by extracting constraints from the program and passing them to a constraint solver, a powerful technique that led to systems such as Stardust (Dunfield 2007) 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 typechecking equality eliminations. In these systems, no algorithm for type inference was given.

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: as long as case analysis cannot modify what is known about an equation, the case analysis is permitted.

In our type system, the checking judgment of the bidirectional algorithm serves to propagate annotations, and our requirement that the scrutinee of a case expression be principal ensures that no equations contain unification variables. This is close in effect to stratified types, and is less expressive than OutsideIn. This is a deliberate design choice to keep the declarative specification simple, rather than an inherent limit of our approach.

To give a specification for the OutsideIn approach, the case rule in our declarative system would be permitted to scrutinize an expression if all types that can be synthesized for it have exactly the same equations, even if they differ in their monotype parts. We feared that such a spec would be much harder for programmers to develop an intuition for than simply saying that a scrutinee must synthesize a unique type. However, the technique we use—higher-order rules with implicational premises like DeclSpineRecover—should work for this case.

More recently, 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.

References

Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic βη-conversion test for Martin-Löf type theory. In *Mathematics of Program Construction (MPC'08)*, volume 5133 of *LNCS*, pages 29–56, 2008.

Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C$^\sharp$. In *OOPSLA*, 2007.

Matthias Blume. 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. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1–3):167–177, 1996.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ICFP*, pages 198–208, 2000.

Joshua Dunfield. Refined typechecking with Stardust. In *Programming Languages meets Programming Verification (PLPV '07)*, 2007.

Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ICFP*, 2013. arXiv:`1306.6032 [cs.PL]`.

Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Found. Software Science and Computation Structures (FOSSACS '03)*, pages 250–266, 2003.

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

Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with GADTs. In *APLAS*, 2013.

Jean-Yves Girard. A fixpoint theorem in linear logic. Post to Linear Logic mailing list, `http://www.seas.upenn.edu/~sweirich/types/archive/1992/msg00030.html`, 1992.

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

Neelakantan R. Krishnaswami. Focusing on pattern matching. In *POPL*, pages 366–378. ACM Press, 2009.

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

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

Dale Miller. Unification under a mixed prefix. *J. Symbolic Computation*, 14(4):321–358, 1992.

Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

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

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

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

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

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

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

Jan M. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *J. Symbolic Logic*, 53(3):840–845, 1988.

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

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*, pages 355–377. Springer LNCS 3085, 2004.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227, 1999.

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

Supplemental material for "Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism and Indexed Types": Complete Rules

This file contains rules omitted in the main paper for space reasons.

Lemmas and proofs are in another, much longer, file.

We also list all the judgment forms:

Judgment	*Description*	*Location*
$\Psi \vdash t : \kappa$	Index term/monotype is well-formed	Figure 14
$\Psi \vdash P\ \mathit{prop}$	Proposition is well-formed	Figure 14
$\Psi \vdash A\ \mathit{type}$	Type is well-formed	Figure 14
$\Psi \vdash \vec{A}\ \mathit{types}$	Type vector is well-formed	Figure 14
$\Psi\ \mathit{ctx}$	Declarative context is well-formed	Figure 14
$\Psi \vdash A \leq^{\pm} B$	Declarative subtyping	Figure 4
$\Psi \vdash P\ \mathit{true}$	Declarative truth	Figure 5
$\Psi \vdash e \Leftarrow A\ p$	Declarative checking	Figure 5
$\Psi \vdash e \Rightarrow A\ p$	Declarative synthesis	Figure 5
$\Psi \vdash s : A\ p \gg C\ q$	Declarative spine typing	Figure 5
$\Psi \vdash s : A\ p \gg C\ \lceil q \rceil$	Declarative spine typing, recovering principality	Figure 5
$\Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p$	Declarative pattern matching	Figure 15
$\Psi \mathrel{/} P \vdash \Pi :: \vec{A} \Leftarrow C\ p$	Declarative proposition assumption	Figure 15
$\Psi \vdash \Pi\ \mathit{covers}\ \vec{A}$	Declarative match coverage	Figure 16
$\Gamma \vdash \tau : \kappa$	Index term/monotype is well-formed	Figure 17
$\Gamma \vdash P\ \mathit{prop}$	Proposition is well-formed	Figure 17
$\Gamma \vdash A\ \mathit{type}$	Polytype is well-formed	Figure 17
$\Gamma\ \mathit{ctx}$	Algorithmic context is well-formed	Figure 17
$[\Gamma]A$	Applying a context, as a substitution, to a type	Figure 9
$\Gamma \vdash P\ \mathit{true} \dashv \Delta$	Check proposition	Figure 18
$\Gamma \mathrel{/} P \dashv \Delta^{\perp}$	Assume proposition	Figure 18
$\Gamma \vdash s \mathrel{\mathring{=}} t : \kappa \dashv \Delta$	Check equation	Figure 19
$s \mathrel{\#} t$	Head constructors clash	Figure 20
$\Gamma \mathrel{/} s \mathrel{\mathring{=}} t : \kappa \dashv \Delta^{\perp}$	Assume/eliminate equation	Figure 21
$\Gamma \vdash A <:^{\pm} B \dashv \Delta$	Algorithmic subtyping	Figure 22
$\Gamma \mathrel{/} P \vdash A <: B \dashv \Delta$	Assume/eliminate proposition	Figure 22
$\Gamma \vdash P \equiv Q \dashv \Delta$	Equivalence of propositions	Figure 22
$\Gamma \vdash A \equiv B \dashv \Delta$	Equivalence of types	Figure 22
$\Gamma \vdash \hat{\alpha} := t : \kappa \dashv \Delta$	Instantiate	Figure 23
$e\ \mathit{chk\text{-}I}$	Checking intro form	Figure 24
$\Gamma \vdash e \Leftarrow A\ p \dashv \Delta$	Algorithmic checking	Figure 12
$\Gamma \vdash e \Rightarrow A\ p \dashv \Delta$	Algorithmic synthesis	Figure 12
$\Gamma \vdash s : A\ p \gg C\ q \dashv \Delta$	Algorithmic spine typing	Figure 12
$\Gamma \vdash s : A\ p \gg C\ \lceil q \rceil \dashv \Delta$	Algorithmic spine typing, recovering principality	Figure 12
$\Gamma \vdash \Pi :: \vec{A} \Leftarrow C\ p \dashv \Delta$	Algorithmic pattern matching	Figure 25
$\Gamma \mathrel{/} P \vdash \Pi :: \vec{A} \Leftarrow C\ p \dashv \Delta$	Algorithmic pattern matching (assumption)	Figure 25
$\Gamma \vdash \Pi\ \mathit{covers}\ \vec{A}$	Algorithmic match coverage	Figure 26
$\Gamma \longrightarrow \Delta$	Context extension	Figure 11
$[\Omega]\Gamma$	Apply complete context	Figure 10

$\boxed{\Psi \vdash t : \kappa}$ Under context Ψ, term t has sort κ

$$\frac{(\alpha : \kappa) \in \Psi}{\Psi \vdash \alpha : \kappa}\ \textsf{UvarSort} \qquad \frac{}{\Psi \vdash 1 : \star}\ \textsf{UnitSort} \qquad \frac{\Psi \vdash t_1 : \star \qquad \Psi \vdash t_2 : \star}{\Psi \vdash t_1 \oplus t_2 : \star}\ \textsf{BinSort}$$

$$\frac{}{\Psi \vdash \mathsf{zero} : \mathbb{N}}\ \textsf{ZeroSort} \qquad \frac{\Psi \vdash t : \mathbb{N}}{\Psi \vdash \mathsf{succ}(t) : \mathbb{N}}\ \textsf{SuccSort}$$

$\boxed{\Psi \vdash P\ \textit{prop}}$ Under context Ψ, proposition P is well-formed

$$\frac{\Psi \vdash t : \mathbb{N} \qquad \Psi \vdash t' : \mathbb{N}}{\Psi \vdash t = t'\ \textit{prop}}\ \textsf{EqDeclProp}$$

$\boxed{\Psi \vdash A\ \textit{type}}$ Under context Ψ, type A is well-formed

$$\frac{(\alpha : \star) \in \Psi}{\Psi \vdash \alpha\ \textit{type}}\ \textsf{DeclUvarWF} \qquad \frac{}{\Psi \vdash 1\ \textit{type}}\ \textsf{DeclUnitWF}$$

$$\frac{\Psi \vdash A\ \textit{type} \qquad \Psi \vdash B\ \textit{type} \qquad \oplus \in \{\rightarrow, \times, +\}}{\Psi \vdash A \oplus B\ \textit{type}}\ \textsf{DeclBinWF}$$

$$\frac{\Psi, \alpha : \kappa \vdash A\ \textit{type}}{\Psi \vdash (\forall \alpha : \kappa.\ A)\ \textit{type}}\ \textsf{DeclAllWF} \qquad \frac{\Psi, \alpha : \kappa \vdash A\ \textit{type}}{\Psi \vdash (\exists \alpha : \kappa.\ A)\ \textit{type}}\ \textsf{DeclExistsWF}$$

$$\frac{\Psi \vdash P\ \textit{prop} \qquad \Psi \vdash A\ \textit{type}}{\Psi \vdash P \supset A\ \textit{type}}\ \textsf{DeclImpliesWF} \qquad \frac{\Psi \vdash P\ \textit{prop} \qquad \Psi \vdash A\ \textit{type}}{\Psi \vdash A \wedge P\ \textit{type}}\ \textsf{DeclWithWF}$$

$\boxed{\Psi \vdash \vec{A}\ \textit{types}}$ Under context Ψ, types in $\vec{A}$ are well-formed

$$\frac{\begin{array}{c}\text{for all } A \in \vec{A}. \\ \Psi \vdash A\ \textit{type}\end{array}}{\Psi \vdash \vec{A}\ \textit{types}}\ \textsf{DeclTypevecWF}$$

$\boxed{\Psi\ \textit{ctx}}$ Declarative context Ψ is well-formed

$$\frac{}{\cdot\ \textit{ctx}}\ \textsf{EmptyDeclCtx} \qquad \frac{\Psi\ \textit{ctx} \qquad x \notin \mathsf{dom}(\Psi) \qquad \Psi \vdash A\ \textit{type}}{\Psi, x : A\ \textit{ctx}}\ \textsf{HypDeclCtx} \qquad \frac{\Psi\ \textit{ctx} \qquad \alpha \notin \mathsf{dom}(\Psi)}{\Psi, \alpha : \kappa\ \textit{ctx}}\ \textsf{VarDeclCtx}$$

Figure 14. Sorting; well-formedness of propositions, types, and contexts in the declarative system

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

$$\frac{}{\Psi \vdash \cdot :: \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatchEmpty} \qquad \frac{\Psi \vdash \pi :: \vec{A} \Leftarrow C\ p \qquad \Psi \vdash \Pi :: \vec{A} \Leftarrow C\ p}{\Psi \vdash \pi \mid \Pi :: \Delta \Leftarrow C\ p}\ \mathsf{DeclMatchSeq} \qquad \frac{\Psi \vdash e \Leftarrow C\ p}{\Psi \vdash (\cdot \Rightarrow e) :: \cdot \Leftarrow C\ p}\ \mathsf{DeclMatchBase}$$

$$\frac{\Psi \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p}{\Psi \vdash (), \vec{\rho} \Rightarrow e :: 1, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatchUnit}$$

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

$$\frac{\Psi \vdash \rho, \vec{\rho} \Rightarrow e :: A_k, \vec{A} \Leftarrow C\ p}{\Psi \vdash \mathsf{inj}_k\ \rho, \vec{\rho} \Rightarrow e :: A_1 + A_2, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatch}{+_k} \qquad \frac{\Psi \,/\, P \vdash \vec{\rho} \Rightarrow e :: A, \vec{A} \Leftarrow C\ p}{\Psi \vdash \vec{\rho} \Rightarrow e :: A \wedge P, \vec{A} \Leftarrow C\ p}\ \mathsf{DeclMatch}\wedge$$

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

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

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

Figure 15. Declarative pattern matching

$\boxed{\Psi \vdash \Pi \textit{ covers } \vec{A}}$ Patterns Π cover the types $\vec{A}$ in context Ψ

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

$$\frac{\Pi \overset{1}{\rightsquigarrow} \Pi' \qquad \Psi \vdash \Pi' \textit{ covers } \vec{A}}{\Psi \vdash \Pi \textit{ covers } 1, \vec{A}}\ \mathsf{DeclCovers1} \qquad \frac{\Pi \overset{\times}{\rightsquigarrow} \Pi' \qquad \Psi \vdash \Pi' \textit{ covers } A_1, A_2, \vec{A}}{\Psi \vdash \Pi \textit{ covers } A_1 \times A_2, \vec{A}}\ \mathsf{DeclCovers}\times$$

$$\frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R \qquad \Psi \vdash \Pi_L \textit{ covers } A_1, \vec{A} \qquad \Psi \vdash \Pi_R \textit{ covers } A_2, \vec{A}}{\Psi \vdash \Pi \textit{ covers } A_1 + A_2, \vec{A}}\ \mathsf{DeclCovers}+ \qquad \frac{\Psi, \alpha : \kappa \vdash \Pi \textit{ covers } \vec{A}}{\Psi \vdash \Pi \textit{ covers } \exists \alpha : \kappa.\ A, \vec{A}}\ \mathsf{DeclCovers}\exists$$

$$\frac{\theta = \mathsf{mgu}(t_1, t_2) \qquad \theta(\Psi) \vdash \theta(\Pi) \textit{ covers } \theta(A_0, \vec{A})}{\Psi \vdash \Pi \textit{ covers } A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{DeclCoversEq} \qquad \frac{\mathsf{mgu}(t_1, t_2) = \bot}{\Psi \vdash \Pi \textit{ covers } A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{DeclCoversEqBot}$$

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

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

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

$$\frac{}{\cdot \overset{+}{\rightsquigarrow} \cdot \parallel \cdot} \qquad \frac{\rho \in \{u, \_\} \qquad \Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{(\rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{+}{\rightsquigarrow} (\_, \vec{\rho} \Rightarrow e) \mid \Pi_L \parallel (\_, \vec{\rho} \Rightarrow e) \mid \Pi_R} \qquad \frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R}{(\mathsf{inj}_1\ \rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{+}{\rightsquigarrow} (\rho, \vec{\rho} \Rightarrow e) \mid \Pi_L \parallel \Pi_R}$$

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

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

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

$$\frac{}{\cdot \overset{\text{var}}{\rightsquigarrow} \cdot} \qquad \frac{\rho \in \{u, \_\} \qquad \Pi \overset{\text{var}}{\rightsquigarrow} \Pi'}{(\rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{\text{var}}{\rightsquigarrow} (\vec{\rho} \Rightarrow e) \mid \Pi'} \qquad \frac{}{\cdot \overset{1}{\rightsquigarrow} \cdot} \qquad \frac{\rho \in \{u, \_, ()\} \qquad \Pi \overset{\text{var}}{\rightsquigarrow} \Pi'}{(\rho, \vec{\rho} \Rightarrow e) \mid \Pi \overset{\text{var}}{\rightsquigarrow} (\vec{\rho} \Rightarrow e) \mid \Pi'}$$

Figure 16. Match coverage

$\boxed{\Gamma \vdash \tau : \kappa}$ Under context Γ, term τ has sort κ

$$\frac{(u : \kappa) \in \Gamma}{\Gamma \vdash u : \kappa}\ \textsf{VarSort} \qquad \frac{(\hat{\alpha} : \kappa = \tau) \in \Gamma}{\Gamma \vdash \hat{\alpha} : \kappa}\ \textsf{SolvedVarSort} \qquad \frac{}{\Gamma \vdash 1 : \star}\ \textsf{UnitSort} \qquad \frac{\Gamma \vdash \tau_1 : \star \qquad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \oplus \tau_2 : \star}\ \textsf{BinSort}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathbb{N}}\ \textsf{ZeroSort} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{succ}(t) : \mathbb{N}}\ \textsf{SuccSort}$$

$\boxed{\Gamma \vdash P \ \textit{prop}}$ Under context Γ, proposition P is well-formed

$$\frac{\Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash t' : \mathbb{N}}{\Gamma \vdash t = t' \ \textit{prop}}\ \textsf{EqProp}$$

$\boxed{\Gamma \vdash A \ \textit{type}}$ Under context Γ, type A is well-formed

$$\frac{(u : \star) \in \Gamma}{\Gamma \vdash u \ \textit{type}}\ \textsf{VarWF} \qquad \frac{(\hat{\alpha} : \star = \tau) \in \Gamma}{\Gamma \vdash \hat{\alpha} \ \textit{type}}\ \textsf{SolvedVarWF} \qquad \frac{}{\Gamma \vdash 1 \ \textit{type}}\ \textsf{UnitWF}$$

$$\frac{\Gamma \vdash A \ \textit{type} \qquad \Gamma \vdash B \ \textit{type} \qquad \oplus \in \{\rightarrow, \times, +\}}{\Gamma \vdash A \oplus B \ \textit{type}}\ \textsf{BinWF} \qquad \frac{\Gamma, \alpha : \kappa \vdash A \ \textit{type}}{\Gamma \vdash \forall \alpha : \kappa.\, A \ \textit{type}}\ \textsf{ForallWF}$$

$$\frac{\Gamma, \alpha : \kappa \vdash A \ \textit{type}}{\Gamma \vdash \exists \alpha : \kappa.\, A \ \textit{type}}\ \textsf{ExistsWF} \qquad \frac{\Gamma \vdash P \ \textit{prop} \qquad \Gamma \vdash A \ \textit{type}}{\Gamma \vdash P \supset A \ \textit{type}}\ \textsf{ImpliesWF} \qquad \frac{\Gamma \vdash P \ \textit{prop} \qquad \Gamma \vdash A \ \textit{type}}{\Gamma \vdash A \wedge P \ \textit{type}}\ \textsf{WithWF}$$

$\boxed{\Gamma \vdash A \ p \ \textit{type}}$ Under context Γ, type A is well-formed and respects principality p

$$\frac{\Gamma \vdash A \ \textit{type} \qquad \mathsf{FEV}([\Gamma]A) = \emptyset}{\Gamma \vdash A \ \mathbf{!} \ \textit{type}}\ \textsf{PrincipalWF} \qquad \frac{\Gamma \vdash A \ \textit{type}}{\Gamma \vdash A \not{\!\!\not{}} \ \textit{type}}\ \textsf{NonPrincipalWF}$$

$\boxed{\Gamma \vdash \vec{A} \ [p] \ \textit{types}}$ Under context Γ, types in $\vec{A}$ are well-formed [with principality p]

$$\frac{\text{for all } A \in \vec{A}. \quad \Gamma \vdash A \ \textit{type}}{\Gamma \vdash \vec{A} \ \textit{types}}\ \textsf{TypevecWF} \qquad \frac{\text{for all } A \in \vec{A}. \quad \Gamma \vdash A \ p \ \textit{type}}{\Gamma \vdash \vec{A} \ p \ \textit{types}}\ \textsf{PrincipalTypevecWF}$$

$\boxed{\Gamma \ \textit{ctx}}$ Algorithmic context Γ is well-formed

$$\frac{}{\cdot \ \textit{ctx}}\ \textsf{EmptyCtx} \qquad \frac{\Gamma \ \textit{ctx} \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash A \ \textit{type}}{\Gamma, x : A \not{\!\!\not{}} \ \textit{ctx}}\ \textsf{HypCtx} \qquad \frac{\Gamma \ \textit{ctx} \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash A \ \textit{type} \qquad \mathsf{FEV}([\Gamma]A) = \emptyset}{\Gamma, x : A \ \mathbf{!} \ \textit{ctx}}\ \textsf{Hyp!Ctx}$$

$$\frac{\Gamma \ \textit{ctx} \qquad u \notin \mathsf{dom}(\Gamma)}{\Gamma, u : \kappa \ \textit{ctx}}\ \textsf{VarCtx} \qquad \frac{\Gamma \ \textit{ctx} \qquad \hat{\alpha} \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash t : \kappa}{\Gamma, \hat{\alpha} : \kappa = t \ \textit{ctx}}\ \textsf{SolvedCtx}$$

$$\frac{\Gamma \ \textit{ctx} \qquad \alpha : \kappa \in \Gamma \qquad (\alpha = -) \notin \Gamma \qquad \Gamma \vdash \tau : \kappa}{\Gamma, \alpha = \tau \ \textit{ctx}}\ \textsf{EqnVarCtx} \qquad \frac{\Gamma \ \textit{ctx} \qquad \blacktriangleright_u \notin \Gamma}{\Gamma, \blacktriangleright_u \ \textit{ctx}}\ \textsf{MarkerCtx}$$

Figure 17. Well-formedness of types and contexts in the algorithmic system

$\boxed{\Gamma \vdash P\ \textit{true} \dashv \Delta}$ Under context Γ, check P, with output context Δ

$$\frac{\Gamma \vdash t_1 \overset{\circ}{=} t_2 : \mathbb{N} \dashv \Delta}{\Gamma \vdash t_1 = t_2\ \textit{true} \dashv \Delta}\ \textsf{CheckpropEq}$$

$\boxed{\Gamma \,/\, P \dashv \Delta^{\perp}}$ Incorporate hypothesis P into Γ, producing Δ or inconsistency $\perp$

$$\frac{\Gamma \,/\, t_1 \overset{\circ}{=} t_2 : \mathbb{N} \dashv \Delta^{\perp}}{\Gamma \,/\, t_1 = t_2 \dashv \Delta^{\perp}}\ \textsf{ElimpropEq}$$

Figure 18. Checking and assuming propositions

$\boxed{\Gamma \vdash t_1 \overset{\circ}{=} t_2 : \kappa \dashv \Delta}$ Check that t_1 equals t_2, taking Γ to Δ

$$\frac{}{\Gamma \vdash u \overset{\circ}{=} u : \kappa \dashv \Gamma}\ \textsf{CheckeqVar} \qquad \frac{}{\Gamma \vdash 1 \overset{\circ}{=} 1 : \star \dashv \Gamma}\ \textsf{CheckeqUnit}$$

$$\frac{\Gamma \vdash \tau_1 \overset{\circ}{=} \tau_1' : \star \dashv \Theta \qquad \Theta \vdash [\Theta]\tau_2 \overset{\circ}{=} [\Theta]\tau_2' : \star \dashv \Delta}{\Gamma \vdash \tau_1 \oplus \tau_2 \overset{\circ}{=} \tau_1' \oplus \tau_2' : \star \dashv \Delta}\ \textsf{CheckeqBin}$$

$$\frac{}{\Gamma \vdash \mathsf{zero} \overset{\circ}{=} \mathsf{zero} : \mathbb{N} \dashv \Gamma}\ \textsf{CheckeqZero} \qquad \frac{\Gamma \vdash t_1 \overset{\circ}{=} t_2 : \mathbb{N} \dashv \Delta}{\Gamma \vdash \mathsf{succ}(t_1) \overset{\circ}{=} \mathsf{succ}(t_2) : \mathbb{N} \dashv \Delta}\ \textsf{CheckeqSucc}$$

$$\frac{\Gamma[\hat{\alpha} : \kappa] \vdash \hat{\alpha} \coloneqq t : \kappa \dashv \Delta \qquad \hat{\alpha} \notin FV(t)}{\Gamma[\hat{\alpha} : \kappa] \vdash \hat{\alpha} \overset{\circ}{=} t : \kappa \dashv \Delta}\ \textsf{CheckeqInstL} \qquad \frac{\Gamma[\hat{\alpha} : \kappa] \vdash \hat{\alpha} \coloneqq t : \kappa \dashv \Delta \qquad \hat{\alpha} \notin FV(t)}{\Gamma[\hat{\alpha} : \kappa] \vdash t \overset{\circ}{=} \hat{\alpha} : \kappa \dashv \Delta}\ \textsf{CheckeqInstR}$$

Figure 19. Checking equations

$\boxed{t_1 \# t_2}$ t_1 and t_2 have incompatible head constructors

$$\frac{}{\mathsf{zero} \# \mathsf{succ}(t)} \qquad \frac{}{\mathsf{succ}(t) \# \mathsf{zero}} \qquad \frac{}{1 \# \tau_1 \oplus \tau_2} \qquad \frac{}{\tau_1 \oplus \tau_2 \# 1} \qquad \frac{\oplus_1 \neq \oplus_2}{\sigma_1 \oplus_1 \tau_1 \# \sigma_2 \oplus_2 \tau_2}$$

Figure 20. Head constructor clash

$\boxed{\Gamma \,/\, \sigma \overset{\circ}{=} \tau : \kappa \dashv \Delta^{\perp}}$ Unify σ and τ, taking Γ to Δ, or to inconsistency $\perp$

$$\frac{}{\Gamma \,/\, \alpha \overset{\circ}{=} \alpha : \kappa \dashv \Gamma}\ \textsf{ElimeqUvarRefl}$$

$$\frac{}{\Gamma \,/\, \mathsf{zero} \overset{\circ}{=} \mathsf{zero} : \mathbb{N} \dashv \Gamma}\ \textsf{ElimeqZero} \qquad \frac{\Gamma \,/\, \sigma \overset{\circ}{=} \tau : \mathbb{N} \dashv \Delta^{\perp}}{\Gamma \,/\, \mathsf{succ}(\sigma) \overset{\circ}{=} \mathsf{succ}(\tau) : \mathbb{N} \dashv \Delta^{\perp}}\ \textsf{ElimeqSucc}$$

$$\frac{\alpha \notin FV(\tau) \qquad (\alpha = -) \notin \Gamma}{\Gamma \,/\, \alpha \overset{\circ}{=} \tau : \kappa \dashv \Gamma, \alpha = \tau}\ \textsf{ElimeqUvarL} \qquad \frac{\alpha \notin FV(\tau) \qquad (\alpha = -) \notin \Gamma}{\Gamma \,/\, \tau \overset{\circ}{=} \alpha : \kappa \dashv \Gamma, \alpha = \tau}\ \textsf{ElimeqUvarR} \qquad \frac{t \neq \alpha \qquad \alpha \in FV(\tau)}{\Gamma \,/\, \alpha \overset{\circ}{=} \tau : \kappa \dashv \perp}\ \textsf{ElimeqUvarL}\perp$$

$$\frac{t \neq \alpha \qquad \alpha \in FV(\tau)}{\Gamma \,/\, \tau \overset{\circ}{=} \alpha : \kappa \dashv \perp}\ \textsf{ElimeqUvarR}\perp$$

$$\frac{}{\Gamma \,/\, 1 \overset{\circ}{=} 1 : \star \dashv \Gamma}\ \textsf{ElimeqUnit} \qquad \frac{\Gamma \,/\, \tau_1 \overset{\circ}{=} \tau_1' : \star \dashv \Theta \qquad \Theta \,/\, [\Theta]\tau_2 \overset{\circ}{=} [\Theta]\tau_2' : \star \dashv \Delta^{\perp}}{\Gamma \,/\, \tau_1 \oplus \tau_2 \overset{\circ}{=} \tau_1' \oplus \tau_2' : \star \dashv \Delta^{\perp}}\ \textsf{ElimeqBin}$$

$$\frac{\Gamma \,/\, \tau_1 \overset{\circ}{=} \tau_1' : \star \dashv \perp}{\Gamma \,/\, \tau_1 \oplus \tau_2 \overset{\circ}{=} \tau_1' \oplus \tau_2' : \star \dashv \perp}\ \textsf{ElimeqBinBot}$$

$$\frac{\sigma \# \tau}{\Gamma \,/\, \sigma \overset{\circ}{=} \tau : \kappa \dashv \perp}\ \textsf{ElimeqClash}$$

Figure 21. Eliminating equations

$\boxed{\Gamma \vdash A <:^{\pm} B \dashv \Delta}$ Under input context Γ, type A is a subtype of B, with output context Δ

$$\dfrac{\begin{array}{l}A \text{ not headed by } \forall/\exists \\ B \text{ not headed by } \forall/\exists \qquad \Gamma \vdash A \equiv B \dashv \Delta\end{array}}{\Gamma \vdash A <:^{\pm} B \dashv \Delta}\;{<:}\mathsf{Equiv}$$

$$\dfrac{\begin{array}{l}B \text{ not headed by } \forall \\ \Gamma, \blacktriangleright_{\hat\alpha}, \hat\alpha : \kappa \vdash [\hat\alpha/\alpha]A <:^{-} B \dashv \Delta, \blacktriangleright_{\hat\alpha}, \Theta\end{array}}{\Gamma \vdash \forall\alpha : \kappa.\, A <:^{-} B \dashv \Delta}\;{<:}\forall\mathsf{L} \qquad \dfrac{\Gamma, \beta : \kappa \vdash A <:^{-} B \dashv \Delta, \beta : \kappa, \Theta}{\Gamma \vdash A <:^{-} \forall\beta : \kappa.\, B \dashv \Delta}\;{<:}\forall\mathsf{R}$$

$$\dfrac{\Gamma, \alpha : \kappa \vdash A <:^{+} B \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash \exists\alpha : \kappa.\, A <:^{+} B \dashv \Delta}\;{<:}\exists\mathsf{L} \qquad \dfrac{\begin{array}{l}A \text{ not headed by } \exists \\ \Gamma, \blacktriangleright_{\hat\beta}, \hat\beta : \kappa \vdash A <:^{+} [\hat\beta/\beta]B \dashv \Delta, \blacktriangleright_{\hat\beta}, \Theta\end{array}}{\Gamma \vdash A <:^{+} \exists\beta : \kappa.\, B \dashv \Delta}\;{<:}\exists\mathsf{R}$$

$$\dfrac{\Gamma \vdash A <:^{-} B \dashv \Delta \qquad \begin{array}{l}\mathit{neg}(A) \\ \mathit{nonpos}(B)\end{array}}{\Gamma \vdash A <:^{+} B \dashv \Delta}\;{<:}^{-}_{+}\mathsf{L} \qquad \dfrac{\Gamma \vdash A <:^{-} B \dashv \Delta \qquad \begin{array}{l}\mathit{nonpos}(A) \\ \mathit{neg}(B)\end{array}}{\Gamma \vdash A <:^{+} B \dashv \Delta}\;{<:}^{-}_{+}\mathsf{R}$$

$$\dfrac{\Gamma \vdash A <:^{+} B \dashv \Delta \qquad \begin{array}{l}\mathit{pos}(A) \\ \mathit{nonneg}(B)\end{array}}{\Gamma \vdash A <:^{-} B \dashv \Delta}\;{<:}^{+}_{-}\mathsf{L} \qquad \dfrac{\Gamma \vdash A <:^{+} B \dashv \Delta \qquad \begin{array}{l}\mathit{nonneg}(A) \\ \mathit{pos}(B)\end{array}}{\Gamma \vdash A <:^{-} B \dashv \Delta}\;{<:}^{+}_{-}\mathsf{R}$$

$\boxed{\Gamma \vdash P \equiv Q \dashv \Delta}$ Under input context Γ, check that P is equivalent to Q with output context Δ

$$\dfrac{\Gamma \vdash t_1 \mathrel{\mathring{=}} t_2 : \mathbb{N} \dashv \Theta \qquad \Theta \vdash [\Theta]t_1' \mathrel{\mathring{=}} [\Theta]t_2' : \mathbb{N} \dashv \Delta}{\Gamma \vdash (t_1 = t_1') \equiv (t_2 = t_2') \dashv \Delta}\;\equiv\!\mathsf{PropEq}$$

$\boxed{\Gamma \vdash A \equiv B \dashv \Delta}$ Under input context Γ, check that A is equivalent to B with output context Δ

$$\dfrac{}{\Gamma \vdash \alpha \equiv \alpha \dashv \Gamma}\;\equiv\!\mathsf{Var} \qquad \dfrac{}{\Gamma \vdash \hat\alpha \equiv \hat\alpha \dashv \Gamma}\;\equiv\!\mathsf{Exvar} \qquad \dfrac{}{\Gamma \vdash 1 \equiv 1 \dashv \Gamma}\;\equiv\!\mathsf{Unit}$$

$$\dfrac{\Gamma \vdash A_1 \equiv B_1 \dashv \Theta \qquad \Theta \vdash [\Theta]A_2 \equiv [\Theta]B_2 \dashv \Delta}{\Gamma \vdash A_1 \oplus A_2 \equiv B_1 \oplus B_2 \dashv \Delta}\;\equiv\!\oplus$$

$$\dfrac{\Gamma, \alpha : \kappa \vdash A \equiv B \dashv \Delta, \alpha : \kappa, \Delta'}{\Gamma \vdash (\forall\alpha : \kappa.\, A) \equiv (\forall\alpha : \kappa.\, B) \dashv \Delta}\;\equiv\!\forall \qquad \dfrac{\Gamma, \alpha : \kappa \vdash A \equiv B \dashv \Delta, \alpha : \kappa, \Delta'}{\Gamma \vdash (\exists\alpha : \kappa.\, A) \equiv (\exists\alpha : \kappa.\, B) \dashv \Delta}\;\equiv\!\exists$$

$$\dfrac{\Gamma \vdash P \equiv Q \dashv \Theta \qquad \Theta \vdash [\Theta]A \equiv [\Theta]B \dashv \Delta}{\Gamma \vdash (P \supset A) \equiv (Q \supset B) \dashv \Delta}\;\equiv\!\supset \qquad \dfrac{\Gamma \vdash P \equiv Q \dashv \Theta \qquad \Theta \vdash [\Theta]A \equiv [\Theta]B \dashv \Delta}{\Gamma \vdash (A \wedge P) \equiv (B \wedge Q) \dashv \Delta}\;\equiv\!\wedge$$

$$\dfrac{\hat\alpha \notin FV(\tau) \qquad \Gamma[\hat\alpha] \vdash \hat\alpha \coloneqq \tau : \star \dashv \Delta}{\Gamma[\hat\alpha] \vdash \hat\alpha \equiv \tau \dashv \Delta}\;\equiv\!\mathsf{InstantiateL} \qquad \dfrac{\hat\alpha \notin FV(\tau) \qquad \Gamma[\hat\alpha] \vdash \hat\alpha \coloneqq \tau : \star \dashv \Delta}{\Gamma[\hat\alpha] \vdash \tau \equiv \hat\alpha \dashv \Delta}\;\equiv\!\mathsf{InstantiateR}$$

Figure 22. Algorithmic equivalence and subtyping

$\boxed{\Gamma \vdash \hat\alpha \coloneqq t : \kappa \dashv \Delta}$ Under input context Γ, instantiate $\hat\alpha$ such that $\hat\alpha = t$ with output context Δ

$$\dfrac{\Gamma_0 \vdash \tau : \kappa}{\Gamma_0, \hat\alpha : \kappa, \Gamma_1 \vdash \hat\alpha \coloneqq \tau : \kappa \dashv \Gamma_0, \hat\alpha : \kappa = \tau, \Gamma_1}\;\mathsf{InstSolve} \qquad \dfrac{\hat\beta \in \mathsf{unsolved}(\Gamma[\hat\alpha : \kappa][\hat\beta : \kappa])}{\Gamma[\hat\alpha : \kappa][\hat\beta : \kappa] \vdash \hat\alpha \coloneqq \hat\beta : \kappa \dashv \Gamma[\hat\alpha : \kappa][\hat\beta : \kappa = \hat\alpha]}\;\mathsf{InstReach}$$

$$\dfrac{\Gamma[\hat\alpha_2 : \star, \hat\alpha_1 : \star, \hat\alpha : \star = \hat\alpha_1 \oplus \hat\alpha_2] \vdash \hat\alpha_1 \coloneqq \tau_1 : \star \dashv \Theta \qquad \Theta \vdash \hat\alpha_2 \coloneqq [\Theta]\tau_2 : \star \dashv \Delta}{\Gamma[\hat\alpha : \star] \vdash \hat\alpha \coloneqq \tau_1 \oplus \tau_2 : \star \dashv \Delta}\;\mathsf{InstBin}$$

$$\dfrac{}{\Gamma[\hat\alpha : \mathbb{N}] \vdash \hat\alpha \coloneqq \mathsf{zero} : \mathbb{N} \dashv \Gamma[\hat\alpha : \mathbb{N} = \mathsf{zero}]}\;\mathsf{InstZero} \qquad \dfrac{\Gamma[\hat\alpha_1 : \mathbb{N}, \hat\alpha : \mathbb{N} = \mathsf{succ}(\hat\alpha_1)] \vdash \hat\alpha_1 \coloneqq t_1 : \mathbb{N} \dashv \Delta}{\Gamma[\hat\alpha : \mathbb{N}] \vdash \hat\alpha \coloneqq \mathsf{succ}(t_1) : \mathbb{N} \dashv \Delta}\;\mathsf{InstSucc}$$

Figure 23. Instantiation

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

$$\frac{}{\lambda x.\, e\ \textit{chk-I}} \qquad \frac{}{()\ \textit{chk-I}} \qquad \frac{}{\langle e_1, e_2\rangle\ \textit{chk-I}} \qquad \frac{}{\mathsf{inj}_k\ e\ \textit{chk-I}}$$

Figure 24. "Checking intro form"

$\boxed{\Gamma \vdash \Pi :: \vec{A} \Leftarrow C\ p \dashv \Delta}$ Under context Γ,
check branches Π with patterns of type $\vec{A}$ and bodies of type C

$$\frac{}{\Gamma \vdash \cdot :: \vec{A} \Leftarrow C\ p \dashv \Gamma}\ \mathsf{MatchEmpty} \qquad \frac{\Gamma \vdash \pi :: \vec{A} \Leftarrow C\ p \dashv \Theta \qquad \Theta \vdash \Pi' :: \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash \pi \mid \Pi' :: \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchSeq}$$

$$\frac{\Gamma \vdash e \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash (\cdot \Rightarrow e) :: \cdot \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchBase} \qquad \frac{\Gamma \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash (), \vec{\rho} \Rightarrow e :: 1, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchUnit}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \vec{\rho} :: A, \vec{A} \Leftarrow C\ p \dashv \Delta, \alpha : \kappa, \Theta}{\Gamma \vdash \vec{\rho} \Rightarrow e :: (\exists \alpha : \kappa.\, A), \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{Match}\exists \qquad \frac{\Gamma \mathrel{/} P \vdash \vec{\rho} \Rightarrow e :: A, \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash \vec{\rho} \Rightarrow e :: A \wedge P, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{Match}\wedge$$

$$\frac{\Gamma \vdash \rho_1, \rho_2, \vec{\rho} \Rightarrow e :: A_1, A_2, \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash \langle \rho_1, \rho_2 \rangle, \vec{\rho} \Rightarrow e :: A_1 \times A_2, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{Match}\times \qquad \frac{\Gamma \vdash \rho, \vec{\rho} \Rightarrow e :: A_k, \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash (\mathsf{inj}_k\ \rho), \vec{\rho} \Rightarrow e :: A_1 + A_2, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{Match}+_k$$

$$\frac{A \text{ not headed by } \wedge \text{ or } \exists \qquad \Gamma, z : A\, \mathbf{!} \vdash \vec{\rho} \Rightarrow e' :: \vec{A} \Leftarrow C\ p \dashv \Delta, z : A\, \mathbf{!}, \Delta'}{\Gamma \vdash z, \vec{\rho} \Rightarrow e :: A, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchNeg}$$

$$\frac{A \text{ not headed by } \wedge \text{ or } \exists \qquad \Gamma \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p \dashv \Delta}{\Gamma \vdash \_, \vec{\rho} \Rightarrow e :: A, \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchWild}$$

$\boxed{\Gamma \mathrel{/} P \vdash \Pi :: \vec{A} \Leftarrow C\ p \dashv \Delta}$ Under context Γ, incorporate proposition P while checking branches Π
with patterns of type $\vec{A}$ and bodies of type C

$$\frac{\Gamma \mathrel{/} \sigma \mathrel{\dot{=}} \tau : \kappa \dashv \bot}{\Gamma \mathrel{/} \sigma = \tau \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p \dashv \Gamma}\ \mathsf{Match}\bot \qquad \frac{\Gamma, \blacktriangleright_P \mathrel{/} \sigma \mathrel{\dot{=}} \tau : \kappa \dashv \Theta \qquad \Theta \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p \dashv \Delta, \blacktriangleright_P, \Delta'}{\Gamma \mathrel{/} \sigma = \tau \vdash \vec{\rho} \Rightarrow e :: \vec{A} \Leftarrow C\ p \dashv \Delta}\ \mathsf{MatchUnify}$$

Figure 25. Algorithmic pattern matching

$\boxed{\Gamma \vdash \Pi\ \textit{covers}\ \vec{A}}$ Under context Γ, patterns Π cover the types $\vec{A}$

$$\frac{}{\Gamma \vdash (\cdot \Rightarrow e_1) \mid \Pi\ \textit{covers}\ \cdot}\ \mathsf{CoversEmpty} \qquad \frac{\Pi \overset{\text{var}}{\rightsquigarrow} \Pi' \qquad \Gamma \vdash \Pi'\ \textit{covers}\ \vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ A, \vec{A}}\ \mathsf{CoversVar} \qquad \frac{\Pi \overset{1}{\rightsquigarrow} \Pi' \qquad \Gamma \vdash \Pi'\ \textit{covers}\ \vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ 1, \vec{A}}\ \mathsf{Covers1}$$

$$\frac{\Pi \overset{\times}{\rightsquigarrow} \Pi' \qquad \Gamma \vdash \Pi'\ \textit{covers}\ A_1, A_2, \vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ A_1 \times A_2, \vec{A}}\ \mathsf{Covers}\times \qquad \frac{\Pi \overset{+}{\rightsquigarrow} \Pi_L \parallel \Pi_R \qquad \Gamma \vdash \Pi_L\ \textit{covers}\ A_1, \vec{A} \qquad \Gamma \vdash \Pi_R\ \textit{covers}\ A_2, \vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ A_1 + A_2, \vec{A}}\ \mathsf{Covers}+$$

$$\frac{\Gamma, \alpha : \kappa \vdash \Pi\ \textit{covers}\ \vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ \exists \alpha : \kappa.\, A, \vec{A}}\ \mathsf{Covers}\exists \qquad \frac{\Gamma \mathrel{/} [\Gamma]t_1 \mathrel{\dot{=}} [\Gamma]t_2 : \kappa \dashv \Delta \qquad \Delta \vdash [\Delta]\Pi\ \textit{covers}\ [\Delta]A_0, [\Delta]\vec{A}}{\Gamma \vdash \Pi\ \textit{covers}\ A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{CoversEq}$$

$$\frac{\Gamma \mathrel{/} [\Gamma]t_1 \mathrel{\dot{=}} [\Gamma]t_2 : \kappa \dashv \bot}{\Gamma \vdash \Pi\ \textit{covers}\ A_0 \wedge (t_1 = t_2), \vec{A}}\ \mathsf{CoversEqBot}$$

Figure 26. Algorithmic match coverage