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 using a focalized subtyping judgment. We use refinement types to infer when a type is principal, leading to reliable 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 indeed use pattern matching to propagate equality information.

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:

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

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

```haskell
left : Sum 0 \to A
left (Left a) = a
```

This definition omits the clause for the `Right` branch. The `Right` branch has index `suc(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 = suc(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 is the identity type of Martin-Löf, but GADTs actually correspond best to the equality of Girard/Schroeder-Heister [1994] and Girard (1992). The Girard/Schroeder-Heister (GSH) approach introduces equality via the reflexivity principle:

\[ \Gamma \vdash t = t \]

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

\[ \text{for all } \theta, \text{ if } \theta \in \text{csu}(s, t) \text{ then } \theta(\Gamma) \vdash \theta(C) \]

Here, we write \( \text{csu}(s, t) \) 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 \( \theta \) 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, the GSH rule is an invertible left rule in the sequent calculus (i.e., the conclusion of the rule implies the premise), which is known to correspond to a pattern matching rule (Krishnaswami 2009). This aligns with the use of GADTs in programming languages like Haskell and Ocaml, which indeed use pattern matching to propagate equality information.

Second, when there are no unifiers, there are no premises: if we assume an inconsistent equation, we can immediately conclude the goal. For example, if we specialize the rule above to the equality \( 0 \equiv 1 \) we get:

\[ \Gamma \vdash (\theta(\Gamma) \equiv C) \]

Together, these two features line up nicely with our definition of `left`, where the impossibility of the `Left` case was indicated by the absence of a pattern clause. So it looks like the use of equality in GADTs corresponds perfectly with the Girard/Schroeder-Heister equality.

Alas, we cannot simply give a proof term assignment for first-order logic and call it a day. The third important feature of the GSH equality rule is its use of unification: it works by treating the free variables of the two terms as unification variables. But type inference algorithms also use unification, introducing unification variables to stand for unknown types. 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 (Laure 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 (§3.5), and sound and complete (Sections §5.6) with respect to the declarative system.

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

Supplementary material. The supplementary material has (1) figures defining all the judgments, including some omitted here for space reasons, and (2) omitted lemma statements and full proofs.

2. Overview
To orient the reader, we give an overview and rationale of the novelties in our type system, before getting into the details of the typing rules and algorithm. We explain our design choices by continuing with the Σ type definition from the introduction. As is well-known (Cheney and Hince 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

\[ \text{Sum } n \triangleq (\text{A } \times \text{ (n = 0)}) + \text{(Zm : N, B } \times \text{ (n = succ(m))}) \]

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 ∀a. a → a, and we want to apply it to the argument 3, then we want to write f 3, and not f [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 ∀a. a → a can be viewed as a subtype of its instantiations (like Nat → 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 Σ 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 λx. x + 1 at type Σa. a → a).

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 \[ \Gamma \vdash A \leq B \text{ for positive types and } \Gamma \vdash A \leq B \text{ for negative types. The positive subtype relation only deconstructs existentials, and the negative subtype relation only deconstructs universals. This fixes the order in which quantifiers are instantiated, making the problem decidable (in fact, rather easy).} \]

The price we pay is that fewer subtype entailments are derivable. But all of the lost subtype entailments are those that rely on “clever” quantifier reversals (which do not arise often in programming); moreover, 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 \texttt{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 pair \((a, b)\) and \((e_1, e_2)\), respectively. 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.

Bidirectionality, pattern matching, and principality. Something that is not, by itself, novel in our approach is our decision to formulate both the declarative and algorithmic systems in a bidirectional style. Bidirectional checking (Pierce and Turner 2000) is a popular implementation choice for systems ranging from dependant 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 not just 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 \texttt{Right} case from the \texttt{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 that term—and its adjoint dual.) So our encoding is really:

\[
\text{sum} \ n \triangleq (A \times \text{unit} + (\exists m : \mathbb{N}. B \times \text{unit}) + \text{unit})
\]

Then \texttt{inj}_1 \( a \) inhabits \text{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).

Expressions

- \( e \triangleq x \mid \varnothing \mid \lambda x : e \mid e_1 \cdot e_2 \mid (e : A) \mid (e_1, e_2) \mid \text{inj}_1 e \mid \text{inj}_2 e \mid \text{case}(e, \Pi) \)

Values

- \( v \triangleq x \mid \varnothing \mid \lambda x : e \mid v_1, v_2 \mid \text{inj}_1 v \mid \text{inj}_2 v \)

Spines

- \( s \triangleq \cdot \mid e - s \)

Patterns

- \( \rho \triangleq x \mid \langle \rho_1, \rho_2 \rangle \mid \text{inj}_1 \rho \mid \text{inj}_2 \rho \)

Branches

- \( \pi \triangleq \rho \Rightarrow e \)

Lists of branches

- \( \Pi \triangleq \cdot \mid (\pi \mid \Pi) \)

### Figure 1. Source syntax

<table>
<thead>
<tr>
<th>Universal variables</th>
<th>( \alpha, \beta, \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorts</td>
<td>( \kappa \triangleq * \mid \mathbb{N} )</td>
</tr>
<tr>
<td>Types</td>
<td>( A, B, C \triangleq A \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 \land P )</td>
</tr>
<tr>
<td>Terms/monotypes</td>
<td>( t, \tau, \sigma \triangleq \text{zero} \mid \text{succ}(t) \mid 1 \mid \alpha \mid \tau \rightarrow \sigma \mid \tau + \sigma \mid \tau \times \sigma )</td>
</tr>
<tr>
<td>Propositions</td>
<td>( P, Q \triangleq \cdot \mid \Pi \mid \Psi, \alpha : \kappa \mid \Psi, x : A \mid p )</td>
</tr>
<tr>
<td>Polaries</td>
<td>( \pm \triangleq - \mid + \mid )</td>
</tr>
<tr>
<td>Binary connectives</td>
<td>( \otimes \triangleq \rightarrow \mid + \mid \times )</td>
</tr>
<tr>
<td>Principality</td>
<td>( p, q \triangleq \cdot \mid \otimes )</td>
</tr>
</tbody>
</table>

Sometimes omitted

### Figure 2. Syntax of declarative types and contexts

- check, eq. elim.
- subtyping
- coverage
- spine typing
- type checking
- match, eq. elim.
- principality recovering
- spine typing
- pattern matching
- type synthesis

### Figure 3. Dependency structure of the declarative judgments

Expression language. Expressions (Figure 1) are variables \( x \), the unit value \( \varnothing \), functions \( \lambda x : e \), applications to spines \( e_1 \cdot e_2 \cdots s \), annotations \( e : A \), pairs \( (e_1, e_2) \), injections into a sum type \( \text{inj}_k e \), and case expressions \( \text{case}(e, \Pi) \) where \( \Pi \) is a list of branches \( \pi \), which can eliminate pairs and injections; see below.
Under context \( \Psi \), type \( A \) is a subtype of \( B \), decomposing head connectives of polarity \( \pm \):

\[
\frac{\Psi \vdash A \; \text{type}}{\Psi \vdash A \leq \pm B} \quad \frac{\Psi \vdash A \leq \pm B}{\Psi \vdash A \leq \pm B} \quad \frac{\Psi \vdash A \leq \pm B}{\Psi \vdash A \leq \pm B}
\]

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

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 \( e \rightarrow s \).

Patterns \( p \) consist of pattern variables, pairs, and injections. A branch \( \pi \) is a sequence of patterns \( \pi \) 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 \( \text{Sorts, terms, monotypes, and propositions.} \)

Terms and monotypes \( t, \tau, \sigma \) share a grammar but are distinguished by their sorts \( k \). Natural numbers zero and succe \( \sigma(1) \) are terms and have sort \( N \). Unit \( 1 \) has the sort \( * \) of monotypes. A variable \( \alpha \) stands for a term or a monotype, depending on the sort \( k \) annotating its binder. Functions, sums, and products of monotypes are monotypes and have sort \( * \). We tend to prefer \( t \) for terms and \( \alpha, \tau \) for monotypes.

A proposition \( P \) or \( Q \) is simply an equation \( t = t' \).

Note that terms, which represent runtime-independent information, are distinct from expressions; however, an expression may include type annotations of the form \( P \supset A \) and \( A \land P \), where \( P \) contains terms.

Contexts. A declarative context \( \Psi \) is an ordered sequence of universal variable declarations \( \alpha : k \) and expression variable typings \( x : A p \), where \( p \) denotes whether the type \( A \) is principal (Section 3.3). A variable \( \alpha \) can be free in a type \( A \) only if \( \alpha \) was declared to the left: \( \alpha : *, x : A p \) is well-formed, but \( x : A p, \alpha : * \) 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 \( \leq \).

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 \( \eta \)-expansions to get programs to typecheck; we omit these rules to keep the implementation complexity low. This also illustrates a nice feature of bidirectional typing: 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 \( A \) \( \leq \) \( A \) means \( A \) is not principal. Note that a principality is part of a judgment, not part of a type. In the checking judgment \( \Psi \vdash e : A \) \( \leq \) \( A \) the type \( A \) is input; if \( p = 0 \), 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 \( \Psi \vdash e \Rightarrow A \), the type \( A \) is output, and \( p = 0 \) 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 \( 0 \) ("not principal"). We write \( p \leq q \) to read "\( p \) at least as principal as \( q \)" for the reflexive closure of \( \leq \).**

Spine judgments. The ordinary form of spine judgment, \( \Psi \vdash s : A p \Rightarrow C q \), says that if arguments \( s \) are passed to a function of type \( A \), the function returns type \( C \). If a function \( e \) applied to one argument \( e_1 \), we write \( e \Rightarrow e_1 \) as syntactic sugar for \( e : (e_1 \Rightarrow \cdots \cdots) \) Supposing \( e \) synthesizes \( A_1 \Rightarrow A_2 \), we apply \( \text{DeclSpine} \) to check \( e_1 \) against \( A_1 \) and using \( \text{DeclEmptySpine} \) to derive \( \Psi \vdash s : A_2 p \Rightarrow A_2 p \).

Rule \( \text{DeclSpine} \) does not decompose \( e \cdot s \) but instantiates a \( \Rightarrow \)-quantifier. Note that, even if the given type \( \forall \alpha : k. A \) is principal \( p = 1 \), the type \( \tau / \alpha A \) in the premise is not principal—we could choose a different \( \tau \). In the fact, the \( q \) in \( \text{DeclSpine} \) is also always \( \Rightarrow \), because no rule deriving the ordinary spine judgment can recover principality.

The recovery spine judgment \( \Psi \vdash s : A_1 \Rightarrow C q \), however, can restore principality in situations where the choice of \( \tau \) in \( \text{DeclSpine} \) cannot affect the result type \( C \). If \( A \) is principal \( p = 1 \) but the ordinary spine judgment produces a non-principal \( C \), we can try to recover principality with \( \text{DeclSpineRecover} \). Its first premise is \( \Psi \vdash s : A_1 \Rightarrow C q \), its second premise (really, an infinite set of premises) quantifies over all derivations of \( \Psi \vdash s : A_1 \Rightarrow C q \). 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 \( q = 1 \) in the conclusion of \( \text{DeclSpineRecover} \).

If some \( C' \neq C \), then \( C \) is certainly not principal, and we must apply \( \text{DeclSpinePass} \) which simply transmits from the ordinary judgment to the recovery judgment.

Figure 3.2 shows the dependencies between the declarative judgments. Given the cycle containing the spine typing judgments, we need to stop and ask: are they well-founded? For well-foundedness of type systems, we can often make a straightforward argument that, as we move from the conclusion of a rule to its premises, either the expression gets smaller, or the expression stays the same but the type gets smaller. In \( \text{DeclSpineRecover} \) neither the expression nor the type get smaller. Fortunately, the rule that gives rise to the arrow from "spine typing" to "type check-
Expression `e` is a checked introduction form

\[ \text{Decl} \Psi \vdash e \iff A \ p \quad \text{Under context } \Psi, \text{ expression } e \text{ checks against input type } A \]

\[ \text{Decl} \Psi \vdash e \Rightarrow A \ p \quad \text{Under context } \Psi, \text{ expression } e \text{ synthesizes output type } A \]

\[ \text{Decl} \Psi \vdash s : A \ p \Rightarrow C \ q \quad \text{Under context } \Psi, \text{ passing spine } s \text{ to a function of type } A \text{ synthesizes type } C; \text{ in the } [q] \text{ form, recover principality in } q \text{ if possible} \]

- **DeclCheckpropEq**
  \[ \Psi \vdash P \text{ true} \quad \text{Under context } \Psi, \text{ check } P \]

- **DeclCheckUnify**
  \[ \Psi \vdash (t = t) \text{ true} \]

\[ \text{DeclCheckpropEq} \]

\[ \Psi \vdash P \text{ true} \quad \text{Under context } \Psi, \text{ check } P \]

\[ \Psi \vdash e \leftrightarrow A ! \quad \text{DeclAnno} \]

\[ \Psi \vdash \{e : A\} \Rightarrow A ! \quad \text{Decl\v{S}pine} \]

\[ \Psi \vdash \tau : \kappa \quad \text{DeclTvSpine} \]

\[ \Psi \vdash e \leftrightarrow s : \{\tau / \kappa \} A \ f \Rightarrow C \ q \quad \text{DeclCSpine} \]

\[ \Psi \vdash e \leftrightarrow A p \quad \text{Decl\rightarrowE} \]

\[ \Psi \vdash s : A \ p \Rightarrow C \ q \quad \text{DeclSpinePass} \]

- **DeclSpineRecov**
  \[ \Psi \vdash s : A ! \Rightarrow C \ q \quad \text{DeclSpineRecov} \]

- **DeclSpinePass**
  \[ \Psi \vdash s : A p \Rightarrow C \ q \quad \text{DeclSpinePass} \]

\[ \Psi \vdash \cdot : A p \Rightarrow A p \quad \text{DeclEmptySpine} \]

\[ \Psi \vdash e \leftrightarrow A k \ p \quad \text{Decl+I} \]

\[ \Psi \vdash \text{inj}_k e \leftrightarrow A_1 k \ p \quad \text{DeclSub} \]

\[ \Psi / P \vdash e \leftrightarrow C p \quad \text{DeclCheck\downarrow} \]

\[ \text{mgue}(\sigma, \tau) = \perp \quad \text{Under context } \Psi, \text{ incorporate proposition } P \]

\[ \text{mgue}(\sigma, \tau) = \emptyset (\Psi) \vdash \emptyset (e) \leftrightarrow \emptyset (C) p \quad \text{DeclCheckUnify} \]

\[ \Psi / (\sigma = \tau) \vdash e \leftrightarrow C p \quad \text{DeclCase} \]

\[ \Psi / (\sigma = \tau) \vdash e \leftrightarrow C p \quad \text{DeclCase} \]

Figure 5. “Checking intro form”

\[ \Psi \vdash \text{inj}_k e \quad \text{inj}_k e \vdash \text{chk-I} \]

\[ \lambda x. \text{e} \quad \text{chk-I} \]

\[ \langle e_1, e_2 \rangle \quad \text{chk-I} \]

\[ \text{inj}_k \ e \quad \text{chk-I} \]

\[ \text{DeclVar} \]

\[ \text{Dec1} \]

\[ \text{Dec1l} \]

\[ \text{Decl/1} \]

\[ \text{DeclSpineRecov} \]

\[ \text{DeclSpinePass} \]

\[ \text{DeclEmptySpine} \]

\[ \text{Decl+I} \]

\[ \text{DeclSub} \]

\[ \text{DeclCheck\downarrow} \]

\[ \text{DeclCheckUnify} \]

\[ \Psi \vdash e \leftrightarrow A p \quad \text{DeclAnno} \]

\[ \Psi \vdash \{e : A\} \Rightarrow A ! \quad \text{DeclAnno} \]

\[ \Psi \vdash \tau : \kappa \quad \text{DeclTvSpine} \]

\[ \Psi \vdash e \leftrightarrow s : \{\tau / \kappa \} A \ f \Rightarrow C \ q \quad \text{DeclCSpine} \]

\[ \Psi \vdash e \leftrightarrow A p \quad \text{Decl\rightarrowE} \]

\[ \Psi \vdash s : A p \Rightarrow C \ q \quad \text{DeclSpinePass} \]

\[ \Psi \vdash \cdot : A p \Rightarrow A p \quad \text{DeclEmptySpine} \]

\[ \Psi \vdash e \leftrightarrow A k \ p \quad \text{Decl+I} \]

\[ \Psi \vdash \text{inj}_k e \leftrightarrow A_1 k \ p \quad \text{DeclSub} \]

\[ \Psi / P \vdash e \leftrightarrow C p \quad \text{DeclCheck\downarrow} \]

\[ \text{mgue}(\sigma, \tau) = \perp \quad \text{Under context } \Psi, \text{ incorporate proposition } P \]

\[ \text{mgue}(\sigma, \tau) = \emptyset (\Psi) \vdash \emptyset (e) \leftrightarrow \emptyset (C) p \quad \text{DeclCheckUnify} \]

\[ \Psi / (\sigma = \tau) \vdash e \leftrightarrow C p \quad \text{DeclCase} \]

**Subtyping.** Rule **DeclSub** involves the subtyping judgment, at the polarity of `B`, the type being checked against. This allows us to play the role of an existential introduction rule, by applying subtyping rules when `B` is an existential type.

**Pattern matching.** Rule **DeclCase** checks that the scrutinee has a principal type, and then invokes the two main judgments for pattern matching. The `Ψ ⊢ Π :: A ≔ C p` judgment checks that each branch in the list of branches `Π` is well-typed, and the `Ψ ⊢ Π covers A` judgment does coverage checking for the list of clauses. Both of these judgments take a vector `A` of pattern types to simplify the specification of coverage checking.

The `Ψ ⊢ Π :: A ≔ C p` judgment (in Figure 7) 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` breaking down a pattern vector headed by a pair pattern `(p, p')`, `p` into `p`, `p'`, `p` (also turning the type sequence from `A × B` into `A, B, 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·` unpacks an existential type, and `DeclMatch/` breaks apart a conjunction by eliminating the equality using unification. The `DeclMatchUnify` 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
Under context $\Psi$, check branches $\Pi$ with patterns of type $\vec{A}$ and bodies of type $C$

$$\vdash \vec{A} \iff C \quad \text{DeclMatchEmpty}$$

$$\vdash \vec{A} \iff C \quad \text{DeclMatchSeq}$$

$$\vdash \vec{A} \iff C \quad \text{DeclMatchBase}$$

$$\vdash \vec{A} \iff C \quad \text{DeclMatchNeg}$$

Figure 7. Declarative pattern matching

Patterns $\Pi$ cover the types $\vec{A}$ in context $\Psi$

$$\vdash (\cdot \Rightarrow e_1) \Pi' \quad \text{DeclCoversEmpty}$$

$$\vdash \Pi \quad \text{DeclCovers1}$$

$$\vdash \Pi \quad \text{DeclCovers2}$$

$$\vdash \Pi \quad \text{DeclCovers3}$$

$$\vdash \Pi \quad \text{DeclCovers4}$$

$$\vdash \Pi \quad \text{DeclCovers5}$$

$$\vdash \Pi \quad \text{DeclCovers6}$$

Figure 8. Match coverage

Remove head variable, wildcard, and unit patterns from $\Pi$

$$\vdash \Pi \quad \text{DeclMatchNeg}$$

$$\vdash \Pi \quad \text{DeclMatchWild}$$

$$\vdash \Pi \quad \text{DeclMatchUnify}$$

$$\vdash \Pi \quad \text{DeclMatchEq}$$
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 \( \mathcal{W} \vdash \Pi \) covers \( \Lambda \) judgment (in Figure 9) 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 \( \Pi \xrightarrow{\Delta} \Pi' \) operation takes every branch \((p, p')\), \( \rho \Rightarrow e \) and expands it to \( p', \rho \Rightarrow e \). To keep the sequence of patterns aligned with the sequence of types, we also expand variables and wildcard patterns into two wildcards: \( x, \beta \Rightarrow e \) becomes \( \gamma, \rho \Rightarrow e \). After expanding out all the pairs, \( \text{DeclCovers} \) checks coverage by breaking down the pair type.

For sum types, we expand a list of branches into two lists, one for each injection. So \( \Pi \xrightarrow{\Delta} \Pi_1 \parallel \Pi_k \) will send all branches headed by \( \text{inj}_1 p \) into \( \Pi_1 \) and all branches headed by \( \text{inj}_2 p \) into \( \Pi_k \), with variables and wildcards being sent to both sides. Then \( \text{DeclCovers} \) can check the coverage of the left and right branches independently.

As with typing, \( \text{DeclCovers} \) just unpacks the existential type. Likewise, \( \text{DeclCoversEqBot} \) and \( \text{DeclCoversEqUniv} \) 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.

These rules do not check for redundancy; \( \text{DeclCoversEmpty} \) applies even when branches are left over. When a \( \top \) is applied, we could mark the \( \cdot \Rightarrow e \) branch, a warning could be issued for any unmarked branches. This seems better as a warning than an error, since redundancy is not stable under substitution. For example, a case with \( \Pi \parallel \Pi \) and \( \text{Cons} \) branches over \( x : \text{Vec } n \), where \( n : \mathbb{N} \), is not redundant—but if we substitute \( 0 \) for \( n \), the \( \text{Cons} \) branch becomes redundant.

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

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 \( \hat{\cdot} \)). 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 \( \Gamma \vdash e : A \mid \Delta \) now takes an input context \( \Gamma \) and yields an output context \( \Delta \) reflecting our increased knowledge of what the types have to be. A judgment \( \Gamma \mid \Delta \) explained in Section 4.1 formalizes the notion of increasing knowledge.

These judgments are documented in Figure 9 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 \( \Gamma \vdash A \equiv B \mid \Delta \) and variable instantiation \( \Gamma \vdash \alpha \equiv t : \kappa \mid \Delta \). Decleratively, 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 14 but rules for most other judgments are in the supplementary material.

Figure 9. Dependency structure of the algorithmic judgments

<table>
<thead>
<tr>
<th>Universal variables</th>
<th>( \alpha, \beta, \gamma )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existential variables</td>
<td>( \hat{\alpha}, \hat{\beta}, \hat{\gamma} )</td>
</tr>
<tr>
<td>Variables</td>
<td>( u ) := ( \alpha</td>
</tr>
<tr>
<td>Types</td>
<td>( A, B, C ) := ( 1</td>
</tr>
<tr>
<td>Propositions</td>
<td>( P, Q ) := ( \top )</td>
</tr>
<tr>
<td>Binary connectives</td>
<td>( \oplus ) := ( \mid + \mid x )</td>
</tr>
<tr>
<td>Terms/monotypes</td>
<td>( t, \tau, \sigma ) := ( \text{zero}</td>
</tr>
<tr>
<td>Contexts</td>
<td>( \Gamma, \Delta, \Theta ) := ( \mid \Gamma, u : \kappa \mid \Gamma, x : A P \mid \Gamma, \hat{\alpha} : \kappa = \tau \mid \Gamma, \alpha = t \mid \Gamma, \Theta )</td>
</tr>
<tr>
<td>Complete contexts</td>
<td>( \Omega ) := ( \mid \Omega, \alpha : \kappa</td>
</tr>
<tr>
<td>Possibly-inconsistent contexts</td>
<td>( \Delta^\perp := \Delta</td>
</tr>
</tbody>
</table>

Figure 10. Syntax of types, contexts, and other objects in the algorithmic system

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

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 10). We use the same meta-variables, e.g. \( A, B, C \) for types. We write \( \bar{u} \) for either a universal variable \( \alpha \) or an existential variable \( \hat{\alpha} \).
More generally, we can apply \( \Omega \) to any context \( \Gamma \) that it extends. This operation of context application \( [\Omega] \Gamma \) is given in Figure 12. The application \( [\Omega] \Gamma \) is defined if and only if \( \Gamma \rightarrow \Omega \) (context extension; see Section 4.4), and applying \( \Omega \) to any such \( \Gamma \) yields the same declarative context \( [\Omega] \Omega \).

Complete contexts are important 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 \( \Gamma \) has the form \( (\Gamma_1, \Theta, \Gamma_2) \). For example, if \( \Gamma = \Gamma_0[\beta] = (\hat{\alpha}, \hat{\beta}, x : \hat{\beta}) \), then \( \Gamma_0[\beta = \hat{\alpha}] = (\hat{\alpha}, \hat{\beta} = \hat{\alpha}, x : \hat{\beta}) \).

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:

\[ \Gamma = \Gamma_0[\theta_1][\theta_2] \]

means \( \Gamma \) has the form \( (\Gamma_1, \Theta_1, \Gamma_2, \Theta_2, \Gamma_3) \).

4.4 The context extension relation

A context \( \Delta \) is extended by a context \( \Delta \), written \( \Gamma \rightarrow \Delta \), if \( \Delta \) has at least as much information as \( \Gamma \), while conforming to the same declarative context—that is, \([\Delta] = [\Omega] \Delta \) for some \( \Omega \).

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

The rules deriving the context extension judgment (Figure 13) say that the empty context extends the empty context \( \rightarrow \); a term variable typing with \( A' \) extends one with \( A \) if applying the extending context \( \Delta \) to \( A \) yields the same type \( [\Delta][A] \); universal variable declarations must match \( [\Delta][\text{Var}] \); equations on universal variables must match \( [\Delta][\text{Eqn}] \); scope markers must match \( [\Delta][\text{Marker}] \); and, existential variables may:

- be unsolved in both contexts \( [\Delta][\text{Unsolved}] \),
- be solved in both contexts, if applying the extending context \( \Delta \) makes the solutions equal \( [\Delta][\text{Solved}] \),
- get solved by the extending context \( [\Delta][\text{Solved}] \),
- be added by the extending context, either without a solution \( [\Delta][\text{Add}] \) or with a solution \( [\Delta][\text{AddSolved}] \).

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

\[ (\hat{\alpha} : \star, \hat{\beta} : \star = \hat{\alpha}) \rightarrow (\hat{\alpha} : \star = 1, \hat{\beta} : \star = \hat{\alpha}) \]

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

\[ (\hat{\alpha} : \star = 1, \hat{\beta} : \star = \hat{\alpha}) \rightarrow (\hat{\alpha} : \star = 1, \hat{\beta} : \star = 1) \]

also holds: while the solution of \( \hat{\beta} \) in \( \Delta \) is different, in the sense that \( \Omega \) contains \( \hat{\beta} : \star = 1 \) while \( \Delta \) contains \( \hat{\beta} : \star = \hat{\alpha} \), applying \( \Omega \) to the two solutions gives the same thing: applying \( \Omega \) to \( \Delta \)'s solution of \( \hat{\beta} \) gives \([\Omega][\hat{\alpha}] = [\Omega][\hat{\alpha}] = 1\), while applying \( \Omega \) to \( \Delta \)'s own solution for \( \hat{\beta} \) also gives 1, because \([\Omega][\hat{\beta}] = 1\).

Extension is quite rigid, however, in two senses. First, if a declaration appears in \( \Gamma \), it appears in all extensions of \( \Gamma \). Second, extension preserves order. For example, if \( \hat{\beta} \) is declared after \( \hat{\alpha} \) in \( \Gamma \), then \( \hat{\beta} \) will also be declared after \( \hat{\alpha} \) in every extension of \( \Gamma \). 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 \( \Gamma, e, A, p \) such that \( \Gamma \vdash e : A \ p \gg C_1 \ q_1 \mid \Delta_1 \) and \( \Gamma \vdash e : A \ p \gg C_2 \ q_2 \mid \Delta_2 \), 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:

**Lemma** (Soundness of Instantiation).
If \( \Gamma \vdash e = \alpha : \kappa \mid \Delta \) and \( \alpha \notin \text{FV}(\Gamma \Delta) \) and \( \Gamma \Delta = \tau \) and \( \Delta \rightarrow \Omega \) then \( |\Omega|\alpha = |\Omega|\tau \).

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

5.2 Elimination lemmas

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

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

1. If \( \Gamma / \sigma \equiv t : \kappa \mid \Delta \) then \( \Delta = (\Gamma, \Theta) \) where \( \Theta = (\alpha_1 = t_1, \ldots, \alpha_n = t_n) \) and for all \( \Omega \) such that \( \Gamma \rightarrow \Omega \) and all \( \tau' \) s.t. \( \Omega \mid \Delta' = \tau' \mid \kappa' \) we have \( |\Omega|\Theta' = (\theta) |\Omega|\tau' \) where \( \theta = \text{mu}(\alpha, t) \).
2. If \( \Gamma / \sigma \equiv t : \kappa \mid \perp \) 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 \( \Gamma \vdash P \text{ true} \mid \Delta \) and \( \Delta \rightarrow \Omega \) then \( \Psi \vdash [\Omega]P \text{ true} \).

**Lemma** (Soundness of Algorithmic Subtyping).
If \( [\Gamma]A = A \) and \( [\Gamma']B = B \) and \( \Gamma \vdash A \text{ type} \) and \( \Gamma \vdash B \text{ type} \) and \( \Gamma \vdash A < : B \mid \Delta \) then \( [\Omega]A \leq_s [\Omega]B \).

**Lemma** (Soundness of Match Coverage).
If \( \Gamma \vdash \Pi \text{ covers } \tilde{A} \) and \( \Gamma \rightarrow \tilde{\Delta} \) and \( \tilde{\Delta}, \tilde{A} \mid \tilde{\Delta} \) and \( \tilde{\Delta} \mid \tilde{\Delta} = \tilde{\Delta} \) then \( \Gamma[(\tilde{\Delta}, \tilde{A})] = \tilde{A} \).

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 \( \Delta \rightarrow \Omega \):

1. If \( \Gamma \vdash e \in A \ p \mid \Delta \) and \( \Gamma \vdash A \ p \mid \Delta \), then \( [\Omega]e \in [\Omega]A \ p \).
2. If \( \Gamma \vdash e \Rightarrow A \ p \mid \Delta \), then \( [\Omega]e \Rightarrow [\Omega]A \ p \).
3. If \( \Gamma \vdash s : A \ p \gg B \ q \mid \Delta \) and \( \Gamma \vdash A \ p \mid \Delta \), then \( [\Omega]s : [\Omega]A \ p \gg [\Omega]B \ q \).
4. If \( \Gamma \vdash s : A \ p \gg B \ q \mid \Delta \) and \( \Gamma \vdash A \ p \mid \Delta \), then \( [\Omega]s : [\Omega]A \ p \gg [\Omega]B \ q \).
5. If \( \Gamma \vdash \Pi : A \ c \mid \Delta \) and \( \Gamma \vdash A \ c \mid \Delta \), then \( [\Omega]\Pi : [\Omega]A \ c \).
6. If \( \Gamma \vdash \Pi : A \ c \mid \Delta \) and \( \Gamma \vdash A \ c \mid \Delta \), then \( [\Omega]\Pi : [\Omega]A \ c \).

Much of this proof is simply “turning the crank”: applying the induction hypothesis to each premise, yielding derivations of corresponding declarative judgments (with \( \Omega \) applied to everything in sight), and applying the corresponding declarative rule; for example, in the **DeclSub** case we finish the proof by applying **DeclSpineRecover**.
We cannot induct on the given derivation alone, because the derivation synthesizes output type \( A \), with output context \( \Delta \).

Under input context \( \Gamma \), expression \( e \) checks against input type \( A \), with output context \( \Delta \).

\[
\Gamma \vdash e \leftrightarrow A \ p \vdash E
\]

Under input context \( \Gamma \), expression \( e \) synthesizes output type \( A \), with output context \( \Delta \).

\[
\Gamma \vdash e \leftrightarrow A \ p \vdash E
\]

We cannot induct on the given derivation alone, because the derivation synthesizes output type \( A \), with output context \( \Delta \).

\[
\Gamma \vdash e \leftrightarrow A \ p \vdash E
\]
full statements of these lemmas, but as an example, if $\Omega[\hat{a} = [\Omega] \tau]$ then $Γ \vdash \hat{a} \models \tau : \kappa \vdash \Delta$, under certain conditions (including that $\hat{a} \notin \text{FV}(\tau)$).

6.1 Separation

To show completeness, we will need to show that wherever the declarative rule $\text{DecSpineRecover}$ is applied, we can apply the algorithmic rule $\text{SpineRecover}$. 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-covering rules are potentially applicable when we start with a principal type $A !$ but produce $C \not\not\not\not$, with $\text{DecSpine}$ changing to $\not\not\not\not$. The proof of completeness (Thm. 3) will use the “all” part of $\text{DecSpineRecover}$ 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 $Γ \vdash s : A ! \array{\Gamma'} \models \Delta$. Since $A !$ is principal, any unsolved existentials in $\Delta$ must have been introduced within this derivation—they can’t be in $Γ$ already. Thus, we might have $\hat{a} : * \vdash s : A ! \models \hat{a} : * \vdash \hat{a} : \hat{b}, \hat{b} : *$ where a $\text{DecSpine}$ subderivation introduced $\hat{b}$, but $\hat{a}$ can’t appear in $\hat{C}$. We also can’t equate $\hat{a}$ and $\hat{b}$ in $\Delta$, which would be morally equivalent to $\Gamma' = \Delta$. Knowing that unsolved existentials in $\hat{C}$ are “new” and independent from those in $Γ$ means we can argue that, if there were an unsolved existential in $\hat{C}$, it would correspond to an unforced choice in a $\text{DecSpine}$ subderivation, invalidating the “for all” part of $\text{DecSpineRecover}$.

Formalizing claims like “must have been introduced” requires several definitions.

Definition 1 (Separation).

An algorithmic context $Γ$ is separable into $Γ_1 * Γ_2$ if (1) $Γ = (Γ_1, Γ_2)$ and (2) for all $∀ \xi : \kappa \vdash \tau \in Γ_1$ it is the case that $\text{FEV}(\tau) \subseteq \text{dom}(Γ_1)$.

If $Γ$ is separable into $Γ_1 * Γ_2$, then $Γ_2$ is self-contained in the sense that all existential variables declared in $Γ_2$ have solutions whose existential variables are themselves declared in $Γ_2$. Every context $Γ$ is separable into $Γ'$ and into $Γ'$.

Definition 2 (Separation-Preserving Extension).

The separated context $Γ_1 * Γ_2$ extends to $Δ_1 * Δ_2$, written $\{Γ_1 * Γ_2\} \to \{\Delta_1 * \Delta_2\}$, if $\{Γ_1, Γ_2\} \to \{\Delta_1, \Delta_2\}$ and $\text{dom}(Γ_1) \subseteq \text{dom}(Δ_1)$ and $\text{dom}(Γ_2) \subseteq \text{dom}(Δ_2)$.

Separation-preserving extension says that variables from one side of $*$ haven’t “jumped” to the other side. Thus, $Δ_1$ may add existential variables to $Γ_1$, but no variable from $Γ_2$ ends up in $Δ_2$ and no variable from $Γ_2$ ends up in $Δ_1$. It is necessary to write $\{Γ_1 * Γ_2\} \to \{\Delta_1 * \Delta_2\}$ rather than $\{Γ_1 \uplus Γ_2\} \to \{Δ_1 \uplus Δ_2\}$, because only $\tau$ includes the domain conditions. For example, $(\hat{a} : \hat{b}) = (\hat{a} : \hat{b}) \models \hat{b}$ has jumped to the left of $*$ in the context $(\hat{a} : \hat{b}) \models \hat{b}$.

We prove many lemmas about separation, but use only one of them in the subsequent development (in the 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 $Γ_2$, then the output context $Δ$ extends $Γ$ and preserves separation, and the output type $C$ mentions only variables in $Δ_2$.

Lemma (Separation—Main).

If $Γ_1 * Γ_2 \vdash s : A p \models \sigma \\uplus Δ_2$ or $Γ_1 * Γ_2 \vdash s : A p \models \sigma \\uplus Δ_2$ and $Γ_1 * Γ_2 \vdash A p \models \text{FEV}(A) \subseteq \text{dom}(Γ_2)$ then $Δ_1 \vdash \Delta_2 \vdash \text{dom}(Γ_2)$ and $\text{dom}(Γ_2)$ and $\text{dom}(C) \subseteq \text{dom}(Δ_2)$.

6.2 Completeness of typing

Theorem 3 (Completeness of Algorithmic Typing).

Given $Γ \vdash \Omega$ such that $\text{dom}(Γ) = \text{dom}(Ω)$:

(i) If $Γ \vdash A p \models [\Omega] e \models [\Omega] A p \models p'$ then there exist $Δ$ and $Ω'$ such that $Δ \vdash Δ$ and $dom(Δ) \subseteq dom(Ω')$ and $Ω' \models \Gamma \models A p' \models A$.

(ii) If $Γ \vdash A p \models [\Omega] e \models [\Omega] A p$ then there exist $Δ$, $Ω'$, $A'$, and $p' \subseteq p$ such that $Δ \vdash Ω'$ and $dom(Δ) = dom(Ω')$ and $Ω' \models A' p' \models A'$.

(iii) If $Γ \vdash A p \models [\Omega] e \models [\Omega] A p \models p'$ then there exist $Δ, Ω'$, $B'$ and $q' \subseteq q$ such that $Δ \vdash Ω'$ and $dom(Δ) = dom(Ω')$ and $Ω' \models A p' \models B' q' \models A' B'$ and $B = [\Omega'] B'$.

(iv) If $Γ \vdash A p \models [\Omega] e \models [\Omega] A p \models B [q]$ and $p' \subseteq p$ then there exist $Δ$, $Ω'$, $B'$, and $q' \subseteq q$ such that $Δ \vdash Ω'$ and $dom(Δ) = dom(Ω')$ and $Ω' \models A p' \models[Ω'] A p' B' q' \models Δ$ and $B' = [Ω'] B'$.

(v) If $Γ \vdash A ! \models [\Omega] e \models [\Omega] A p \models B [q]$ and $p' \subseteq p$ then there exist $Δ$, $Ω'$, and $C$ such that $Δ \models Ω'$ and $dom(Δ) = dom(Ω')$ and $Ω' \models A p' \models [Ω'] A p' B' q' \models Δ$ and $B' = [Ω'] B'$.

(vi) If $Γ \vdash A ! \models [\Omega] e \models [\Omega] A p \models B [q]$ and $p' \subseteq p$ then there exist $Δ$, $Ω'$, and $C$ such that $Δ \models Ω'$ and $dom(Δ) = dom(Ω')$ and $Ω' \models A p' \models [Ω'] A p' B' q' \models Δ$ and $B' = [Ω'] B'$.

Proof sketch—$\text{DecSpineRecover}$ case. By i.h., $Γ \vdash s : [Γ] A ! \models \Gamma' \vdash \Gamma' \models A ! \array{\Gamma'} \models \Delta$ where $Δ \vdash Ω'$ and $Ω' \models Δ$ and $dom(Δ) = dom(Ω')$ and $C = [\Omega'] C'$. To apply $\text{SpineRecover}$ we need to show $\text{FEV}(\hat{C}) = \emptyset$. Suppose, for a contradiction, that $\text{FEV}(\hat{C}) \neq \emptyset$. Construct a variant of $\hat{C}$ called $Ω_2$ that has a different solution for some $\hat{a} \notin \text{FEV}(\hat{C})$. By soundness (Thm. 3), $Ω_2 \models [\Omega_2] A ! \models [\Omega_2] C'$. Using the separation lemma with the trivial separation $Γ = (Γ' * s)$ we get $Δ' = (Δ_1 * Δ_2)$ and $Γ' \models Δ_2$ and $\text{FEV}(C') \subseteq \text{dom}(Δ_2)$. 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 $Γ \models Ω_2 \models [Ω_2] e : [Ω_2] ! \models [Ω_2] C'$. Now instantiate the “for all $C'$” premise with $C_2 = [Ω_2] C'$, giving $C = [Ω_2] C'$. But we chose $Ω_2$ to have a different solution for $\hat{a} \notin \text{FEV}(C)$, so we have $C \neq [Ω_2] C'$. Contradiction. Therefore $\text{FEV}(\hat{C}) = \emptyset$, so we can apply $\text{SpineRecover}$.

7. Extensions

To keep our type systems manageable, we left out some features that would be desirable, if not essential, in practice. Here, we give some examples that use some of these features. To model Haskell or ML-style datatype declarations, the key extensions are (1) type constructors that take arguments and (2) recursive types (Pierce 2002 chapter 20). These extensions appear relatively straightforward, especially if we accept weak subtyping (if we have no subtyping for products, we may as well have no subtyping for e.g. list elements).

Lists with length. We can define a GADT of lists that track their length:

List α : (n : N) ∉ (1 ∧ (n=0)) + (3 m N. α × List α m ∧ (n=succ(m)))

Black height. Red-black trees (Okasaki 1998) are a nice application of indexed types, with an index tracking the black height—the number of black nodes on every path to a leaf. Here, we use a three-way sum whose first component represents an empty (leaf) node,
whose second component represents a red branch node, and whose third component represents a black branch node:

\[
\text{Rbt}\ \alpha\ (h : N) \triangleq (\lambda x. (h = 0) + (\alpha \times \text{Rbt}\ \alpha x) \times \text{Rbt}\ \alpha y) + (\exists h'. N. (\alpha \times \text{Rbt}\ \alpha h' \times \text{Rbt}\ \alpha h')) \wedge (h' = \text{succ}(h))
\]

In these encodings, different data constructors do not yield different return types, as in the surface syntax of Haskell and OCaml. Instead, we constrain the parameter using equality constraints. So \text{Lie}\ \alpha \ 0 \ only \ has \ the \ left \ branch \ of \ the \ sum \ (i.e., \ the \ nil \ case) \ as \ an \ inhabitant, \ since \ there \ is \ no \ way \ to \ prove \ the \ equality \ required \ by \ the \ right \ branch \ (\emptyset \ is \ not \ the \ successor \ of \ anything).

The main issue is that both of these definitions use existentials and other “large” type connectives, and our system seemingly relies on monotypes (which cannot contain such connectives): polymorphic functions can only be instantiated with monotypes.

This limitation should create no difficulties in typical practice, if we treat user-defined type constructors, such as \text{Lie}, as monotypes (provided all their arguments are monotypes), and expand the definition only as needed: when checking an expression against a user type constructor, and when demanded by pattern matching. When an expression of existential type is pattern-matched, the existential enters the context, along with the equation carried by the \wedge connective. For example, if we pattern-match an expression of type \text{Lie}\ \alpha 5\ and \ are \ inside \ a \ branch \ whose \ pattern \ is \ \text{inj}_2 (y, ys), \ the \ variable \ ys \ will \ have \ type \ \text{Lie}\ \alpha m \ where \ m = 4.

To define subtyping for user-defined type constructors, we would probably need variance annotations stating whether arguments like \alpha \ are co-, contra-, or bi-variant. For other operations, such as context application, user type constructors could be treated similarly to sums and products.

8. 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 4 there are two logical accounts of equality—the identity type of Martin-Löf and the equality type of Schroeder-Heister (1984) and Girard (1982). 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_1-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 intractability 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 Rémy 2003) 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 \text{DeciSpineRecover}—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 \beta_1-conversion test for Martin-Löf type theory. In

2013/7/9


Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types.


