Boxy Types: Inference for Higher-Rank Types and Impredicativity

Dimitrios Vytiniotis Stephanie Weirich
University of Pennsylvania
{dimitriv,sweirich}@cis.upenn.edu

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

Languages with rich type systems are beginning to employ a blend of type *inference* and type *checking*, so that the type inference engine is guided by programmer-supplied type annotations. In this paper we show, for the first time, how to combine the virtues of two well-established ideas: unification-based inference, and bidirectional propagation of type annotations. The result is a type system that conservatively extends Hindley-Milner, and yet supports both higher-rank types and impredicativity.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—abstract data types, polymorphism

General Terms Languages, Theory

Keywords impredicativity, higher-rank types, type inference

1. Introduction

Consider the following function:

```
f get = (get [1,2], get ['a','b','c'])
```

The argument get is applied to a list of Int and to a list of Char, so get would have to be polymorphic for this program to type-check. The Hindley-Milner type system [13] rejects this program because it requires that function arguments have monomorphic types. However, suppose we add a type annotation:

```
f :: (forall a. [a] -> a) -> (Int, Char)
f get = (get [1,2], get ['a','b','c'])
```

Then it is plain as a pikestaff what polymorphic type to attribute to get! This observation leads to the following simple idea: *exploit type annotations to guide a simple inference algorithm to find types for programs that would be untypeable by Hindley-Milner*.

We have two particular extensions in mind. The first is *higher-rank types*; that is, types with $\forall$ quantifiers nested inside function types. They have already proved to be extremely useful in practice: not many programs need them, but when they are needed they are absolutely indispensable (e.g. [8, 22]).

However, higher-rank types are not enough! Consider the following variation:

```
g :: Maybe (forall a. [a] -> a) -> (Int, Char)
g Nothing    = (0,          '0')
g (Just get) = (get [1,2], get ['a','b','c'])
```

Here, the polymorphic function is wrapped in a Maybe type, an ordinary algebraic data type whose declaration looks like this:

```
data Maybe a = Nothing | Just a
```

In the Hindley-Milner system, type variables range only over *monotypes*, but to allow function g we must allow a type variable—a, in the definition of Maybe—to be instantiated with a *polytype*. In the jargon, we need an *impredicative* type system, whereas Hindley-Milner is *predicative*.

In this paper we show how to support both higher-rank types and impredicativity, using programmer type annotations to guide unification-based type inference. Specifically:

- We give a type system for source programs that include type annotations, higher rank types, and impredicativity (Section 3). Its key innovation is the notion of a *boxy type*, which expresses the direction of information flow in an inference algorithm.
- Our type system is a conservative extension of the Hindley-Milner type system (Section 4.5). It is also expressive: any System F program can be written in our language, through the addition of type annotations (Section 4.3).
- The type system has a relatively simple inference algorithm, namely a modest extension of the well-known Damas-Milner type inference algorithm (Section 5). We have proven that this algorithm is sound and complete.
- To significantly reduce the burden of type annotations for practical programming, we present a refinement of our basic type system, in Section 6.

We believe that this paper is the first to combine the virtues of unification-based type inference with bidirectional propagation of known types. The idea of combining the two has been around in the folklore for a few years, but formalising it is tricky, and constitutes our main contribution. We discuss related work in Section 8.

We have concrete evidence that our inference algorithm is, as claimed, a modest extension of the conventional Damas-Milner algorithm: we have fully implemented it in the Glasgow Haskell Compiler (GHC), the world's leading Haskell compiler. The changes required were modest and non-invasive, even though GHC already embodies an extremely complicated type system that includes higher-kinded type variables, multi-parameter type classes, functional dependencies, overlapping instances, implicit parameters, template meta-programming, and more. In particular, we found no difficulty in combining the boxy types of this paper with the wobbly types of our companion paper about type inference for GADTs [18].

2. The key ideas of this paper

In this section we briefly introduce the main ideas of the type system. Section 3 follows immediately, and fills in the missing details.

Our "gold standard" for expressiveness is System F, whose types have the form:

$$\sigma \quad ::= \quad a \mid \sigma_1 \to \sigma_2 \mid \forall a\,.\,\sigma \mid \texttt{[}\sigma_1\texttt{]} \mid \texttt{Int}$$

Here we extend classical System F with list and integer types, the former being written with square brackets. Unlike the Hindley-Milner system, a $\forall$ quantifier can be nested inside a function type. Furthermore, in System F a polymorphic function can be called at a polymorphic type. For example, if

$$\begin{array}{rcl} \texttt{length} & :: & \forall a.\texttt{[}a\texttt{]} \to \texttt{Int} \\ \texttt{ids} & :: & \texttt{[}\forall c.\, c \to c\texttt{]} \end{array}$$

then the term $(\texttt{length}\ (\forall c.\, c \to c)\ \texttt{ids})$ instantiates `length` at the polymorphic type $\forall c.\, c \to c$. This, too, is not permitted by the Hindley-Milner system.

Why does Hindley-Milner prohibit these constructs? Because the System F is an *explicitly-typed* language, whereas Hindley-Milner is *implicitly typed*. Specifically, in System F (i) every binder is annotated with its type and (ii) type abstractions and applications are explicit. In contrast, in the Hindley-Milner type system, binders are not required to be typed, and type abstractions and applications are simply omitted. One can regard type inference as the process of inferring the missing annotations, type abstractions and type applications.

2.1 Higher rank types

The restrictions of the Hindley-Milner system make type inference tractable. Recall the example from the Introduction:

```
f get = (get [1,2], get ['a','b','c'])
```

Without assistance, it is hard to figure out what type to attribute to `get`. Should it be $\forall a.\texttt{[}a\texttt{]} \to a$? Or $\forall a.\texttt{[}a\texttt{]} \to \texttt{Int}$? Or some other type? There is no most general type, so we cannot perform modular type inference; that is, we cannot infer a single type for `f` to use throughout its scope. Choosing either of the above types for the argument is sound, but once we commit to one we cannot apply `f` to arguments with the other type.

It is very hard to "guess" a suitable type for `get` by looking at its occurrences. Yet, that is exactly how unification-based type inference works—but it *only* works because the "guessed" types, such as function arguments, are all *monotypes*. This observation is so important that we elevate it to a design principle:

Principle 1 (Guessing) *Type inference can "guess" a monotype, but should not "guess" a polytype.*

There are more sophisticated type-inference algorithms that can, in some cases, guess polytypes (e.g. [7]). However, if we can arrange that only monotypes need to be guessed, then a relatively simple type inference algorithm might—indeed, does—suffice.

In particular, the need for "guessing" is obviated if the programmer supplies a type signature:

```
f :: (forall a. [a] -> a) -> (Int, Char)
f get = (get [1,2], get ['a','b','c'])
```

Now, the problem reduces to type *checking*. The programmer has, in effect, supplied the type for `get`, and that allows us to check the body of `f` without difficulty. That suggests our first main (well-known) technique:

Idea 1: *propagate contextual type information inwards, to give polymorphic types to binders.*

The standard way to express this idea is called *local type inference* [19]. In local type inference there are two judgements forms instead of one: $\Gamma \vdash_{\Uparrow} t : \tau$ and $\Gamma \vdash_{\Downarrow} t : \tau$. The former means "infer the type of t", while the latter means "check that t has the known type τ". Operationally, we think of τ as an output of the first judgement, but as an input to the second.

The separation between checking mode and inference modes allows us to assign higher-rank types to functions while still satisfying Principle 1. The idea is that the bound variable of a lambda can be given a polytype if we are in checking mode, but only a monotype in inference mode:

$$\frac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} t : \sigma_2}{\Gamma \vdash_{\Downarrow} (\backslash x\ .\ t) : \sigma_1 \to \sigma_2}\ \text{ABS1} \qquad \frac{\Gamma, x : \tau \vdash_{\Uparrow} t : \sigma}{\Gamma \vdash_{\Uparrow} (\backslash x\ .\ t) : \tau \to \sigma}\ \text{ABS2}$$

Reading the ABS2 rule from bottom to top, to infer the type of a lambda expression, we need to guess the argument type τ to add to the context and infer the type of the body. However, in ABS1, because we are checking that the lambda expression has a function type, we do not need to guess what type to add to the environment.

During type inference, where do the input types for checking mode come from? One source is from programmer annotations. If the language allows the programmer to annotate term t with a specified type τ, $(t::\tau)$, then we could have the following rule that switches between inference and checking modes.

$$\frac{\Gamma \vdash_{\Downarrow} t : \tau}{\Gamma \vdash_{\Uparrow} (t::\tau) : \tau}\ \text{SIG}$$

Furthermore function applications are another rich source of contextual type information. For example, if

$$\texttt{f} : ((\forall a.\, a \to a) \to \texttt{Int}) \to \texttt{Int}$$

then in the application $(\texttt{f}\ (\backslash\texttt{x}.t))$, we should be able to give `x` the polytype $\forall a.\, a \to a$, in t.

2.2 Impredicativity

A similar line of reasoning applies to impredicativity. In Haskell, programmers do not write type applications, so the type arguments of a polymorphic function must be inferred. Consider again the function `length` of type $\forall a\ .\ \texttt{[}a\texttt{]} \to \texttt{Int}$. How can we figure out the type argument of `length` in the application `(length ids)`? The conventional Damas-Milner algorithm instantiates `length`'s type with fresh type variables—these are the guessed types—and then uses unification to elaborate them. But, as we have mentioned, unification should not guess a polytype.

In this case, though, a much easier approach suggests itself. Given that `ids` has type $\texttt{[}\forall c.\, c \to c\texttt{]}$, what instantiation of a will make the type $\texttt{[}a\texttt{]}$ match? Easy: instantiate a to $\forall c.\, c \to c$! So the second idea is:

Idea 2: *use locally known information at a function call to instantiate a polymorphic function.*

2.3 Partial type information

There is a sharp tension between Idea 1 and Idea 2. The former suggested a rule for typing applications which *inferred* the function (instantiating it in the process) and *checked* the type of the argument. But Idea 2 suggests instead that we should *infer* the type of the argument, so that we know how to instantiate the type of the function.

— Terms —

$$\begin{array}{lcl} t, u & ::= & \nu \mid \backslash x . t \mid t\ u \\ & \mid & \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t \\ & \mid & \texttt{let}\ x\texttt{::}\sigma = u\ \texttt{in}\ t \\ \nu & ::= & x \mid C \end{array}$$

— Vanilla types —

$$\begin{array}{lcl} \tau & ::= & a \mid \tau_1 \rightarrow \tau_2 \mid T\ \overline{\tau} \\ \rho & ::= & \tau \mid \sigma \rightarrow \sigma \mid T\ \overline{\sigma} \\ \sigma & ::= & \forall \overline{a} . \rho \end{array}$$

— Boxy types —

$$\begin{array}{lcl} \rho' & ::= & \tau \mid \sigma' \rightarrow \sigma' \mid T\ \overline{\sigma}' \mid \boxed{\rho} \\ \sigma' & ::= & \forall \overline{a} . \rho' \mid \boxed{\sigma} \end{array}$$

— Environments —

$$\Gamma \ ::= \ \cdot \mid \Gamma, a \mid \Gamma, (x{:}\sigma)$$

Figure 1: Syntax of the source language and types

But this tension is more apparent than real. Consider again our example `(length ids)`. Intuitively, we would like to type check it in the following way:

1. Examine the type of `length` : $\forall a\ .\ \texttt{[}a\texttt{]} \rightarrow \texttt{Int}$.
2. Check the type of the argument, expecting it to have type [*something*]. The list part of this type—denoted by the brackets—is determined by the function; but the "*something*" is discovered from the argument itself.
3. Use the "*something*" to instantiate the type of `length`.

This approach suggests the third main idea:

Idea 3: *combine the two judgement forms,* $\vdash_{\Downarrow}$ *and* $\vdash_{\Uparrow}$*, into a single judgement* $\Gamma \vdash t : \tau'$ where τ' is like τ except that it may contain "holes" that correspond to the inferred part of the type.

We denote these "holes" using boxes. For example:

$$\Gamma \vdash t : \boxed{\texttt{Bool}} \rightarrow \texttt{Int}$$

This judgement *checks* that t has a type of form *something* $\rightarrow$ `Int`; and *infers* that the *something* is `Bool`. The earlier inference and checking judgements are now just special cases:

$$\begin{array}{lll} \Gamma \vdash_{\Downarrow} t : \tau & \text{is written} & \Gamma \vdash t : \tau \\ \Gamma \vdash_{\Uparrow} t : \tau & \text{is written} & \Gamma \vdash t : \boxed{\tau} \end{array}$$

Beyond resolving the tension between Ideas 1 and 2, boxes allow us to type more programs than with bidirectional judgments—we will see an example when we discuss the application rule in Section 3.2. However, although boxes appear in the structure of types, we do not mean for them to have any *logical* interpretation, such as under the Curry-Howard isomorphism. Instead, their purpose is to make sure that all derivations in the specification of our type system have a corresponding algorithmic interpretation.

The idea of using the structure of a type, rather than a judgement, to describe information flow is not new. Odersky, Zenger & Zenger [16] coined the term "Colored Local Type Inference" to describe a type inference system for $F_{\leq}$ that uses red and blue colours to describe which parts of the type are checked and which are inferred. Although the motivating ideas are very similar, the details are entirely different, as we discuss in Section 8.

3. Type system specification

We next present the specification of our type system. Although its design was guided by the practicalities of our type inference algorithm, and its rules are directed by the syntax of the term. It is not an algorithm and leaves details such as first-order unification implicit. We describe the inference algorithm in Section 5.

3.1 Syntax

The syntax of our source language, and of its types, is given in Figure 1. The language is just large enough to demonstrate all the interesting features of the type system. It includes type constructors, denoted with T, and data constructors, denoted with C. Other familiar features, such as recursive `let` and pattern-matching, could be easily added.

The source language allows the programmer to annotate a let binding with a type σ. This annotation need not be closed—it may refer to type variables in the enclosing context. Thus we support a form of lexically-scoped type variables [23], an extension that is essential to the expressiveness of our language, but otherwise does not interact with our type system. We defer the details of the lexically-scoped type variables to Section 3.3.

Types come in two forms. *Vanilla types* are stratified into *monotypes*, τ, and *polytypes*, ρ and σ. Monotypes have no $\forall$'s anywhere, whereas polytypes may have $\forall$'s. A ρ-type has no $\forall$'s at the top, but may contain polytypes. The notation $\overline{a}$ denotes a sequence of zero or more distinct type variables. We often implicitly coerce between sequences and sets in the usual manner.

A *boxy type*, written with a prime (ρ', σ'), is a type containing zero or more boxes, each box containing a vanilla type (Figure 1). Boxy types ρ' and σ' are stratified in exactly the same way as their vanilla counterparts. However note that *inside a box is a vanilla type*; that is, boxes are not nested. The intuition here is that a boxy type has an outer structure of "checked" information, which switches at the boxes to an inner structure of "inferred" information.

Boxy types also satisfy a non-syntactic invariant: *in a type* $\forall \overline{a} . \rho'$*, none of the quantified variables* $\overline{a}$ *may appear inside a box in* ρ'. This invariant allows us to instantiate a polymorphic type $\forall \overline{a} . \rho'$ with boxy types $\overline{\sigma'}$, and still obtain a syntactically well-formed type $[\overline{a \mapsto \sigma'}]\rho'$. In the absence of the invariant, the instantiated type might have nested boxes. This invariant is maintained by our typing rules.

The type inside a box may, nevertheless, have *free* type variables. For example, the rules validate the following:

$$\frac{\Gamma \vdash u : \boxed{a \rightarrow a} \quad \Gamma, x : \forall a . a \rightarrow a \vdash t : \rho'}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho'}\ \text{LET-I}$$

Here, the type of u is $\boxed{a \rightarrow a}$, in which a appears free inside the box. Intuitively, we can think of this rule as performing two steps: First, the type of u is inferred to be $\boxed{a \rightarrow a}$. Second, the type $a \rightarrow a$ is extracted from the box, generalized, and then placed in the context when we check t. It is an invariant of our algorithm that all boxes will be filled in after type checking. Therefore, the type $a \rightarrow a$ is known type information after looking at u.

Environments Γ attribute vanilla types to term variables and also bind lexically-scoped type variables. We use the notation $ftv(\sigma')$ to refer to the free type variables of a type (both in and out of boxes) and we extend this notation to environments, so that $ftv(\Gamma)$ refers to all the free variables appearing anywhere in the types of the environment, including the bindings for lexical type variables. We use the notation $dom(\Gamma)$ for the collection of term and type variables bound in Γ. We write $\#$ for the binary set-disjointness operator.

The notation $\overline{\sigma'}$ denotes sequences of σ'-types, not necessarily distinct. We write $[\overline{a \mapsto \sigma'}]$ for the capture-avoiding substitution with domain the set $\overline{a}$ that maps each $a \in \overline{a}$ to the corresponding type $\sigma' \in \overline{\sigma'}$. We often abbreviate $\forall \emptyset . \rho'$ as ρ' (and similarly for vanilla types).

$$\boxed{\Gamma \vdash t : \rho'}$$

$$\frac{\nu{:}\sigma \in \Gamma \quad \vdash \sigma \leq \rho'}{\Gamma \vdash \nu : \rho'}\ \textsc{var} \qquad \frac{\begin{array}{c}\Gamma \vdash t : \boxed{\sigma} \to \rho' \\ \Gamma \vdash^{poly} u : \sigma\end{array}}{\Gamma \vdash t\ u : \rho'}\ \textsc{app}$$

$$\frac{\begin{array}{c}\vdash \sigma_1' \sim \boxed{\sigma_1} \\ \Gamma, x{:}\sigma_1 \vdash^{poly} t : \sigma_2'\end{array}}{\Gamma \vdash (\backslash x\,.\,t) : \sigma_1' \to \sigma_2'}\ \textsc{abs}1 \qquad \frac{\Gamma \vdash (\backslash x\,.\,t) : \boxed{\sigma_1} \to \boxed{\sigma_2}}{\Gamma \vdash (\backslash x\,.\,t) : \boxed{\sigma_1 \to \sigma_2}}\ \textsc{abs}2$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \boxed{\rho} \\ \overline{a} = \mathit{ftv}(\rho) - \mathit{ftv}(\Gamma) \\ \Gamma, x{:}\forall\overline{a}\,.\,\rho \vdash t : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{=}\ u\ \mathtt{in}\ t : \rho'}\ \textsc{let-i}$$

$$\frac{\begin{array}{c}\mathit{ftv}(\forall\overline{a}\,.\,\rho) \subseteq \mathit{dom}(\Gamma) \quad \overline{a}\#\mathit{ftv}(\Gamma) \\ \Gamma, \overline{a} \vdash u : \rho \\ \Gamma, x{:}\forall\overline{a}\,.\,\rho \vdash t : \rho'\end{array}}{\Gamma \vdash \mathtt{let}\ x\mathtt{::}\forall\overline{a}\,.\,\rho\ \mathtt{=}\ u\ \mathtt{in}\ t : \rho'}\ \textsc{let-s}$$

$$\boxed{\Gamma \vdash^{poly} t : \sigma'}$$

$$\frac{\Gamma \vdash t : \rho' \qquad \overline{a}\#\mathit{ftv}(\Gamma) \quad \overline{a} \text{ not inside boxes in } \rho'}{\Gamma \vdash^{poly} t : \forall\overline{a}\,.\,\rho'}\ \textsc{gen}$$

Figure 2: Type system specification

3.2 The core rules

The main type system is given in Figure 2. The main judgement has the familiar form

$$\Gamma \vdash t : \rho'$$

However, notice that the type attributed to t is a *boxy* type ρ'. Our operational understanding of the judgement is that we type check t, *checking* ρ' *outside* boxes, and *inferring inside* boxes. The boxes are like **var** parameters in Pascal: they are the "holes" in which the results are returned.

Because an environment Γ attributes a vanilla type σ to each variable x that it binds, *the only place that boxes may appear is in the result type of the judgement*. One could also imagine allowing boxy types in Γ, as we discuss in Section 7.

The rules for this type system are syntax-directed, with type instantiation being inferred at variable occurrences and generalisation at let bindings.[1] Specifically, rule VAR deals with variable or data constructor occurrences by invoking an instantiation relation $\vdash \sigma \leq \rho'$, which we discuss in Section 3.4. The rule for unannotated let expressions (LET-I) is completely straightforward: it infers the type of the right-hand side u, generalises it in the usual Hindley-Milner way, and type checks the body t.

Now consider the rules for lambda abstractions. Rule ABS1 has a familiar form, except for two points. First, we must use an auxiliary judgement $\vdash^{poly}$ (also in Figure 2) to type check the body of the abstraction because the type to the right of the arrow, σ_2', is not necessarily a ρ'-type. In rule GEN note that in pure inference mode — when there is a box around the complete result type — we do no generalisation. Second, in ABS1 one might be tempted to extend Γ with the boxy typing $x{:}\sigma'$, but that would invalidate our invariant that Γ contains only vanilla types. The premise $\vdash \sigma_1' \sim \boxed{\sigma_1}$ finds a vanilla type σ_1 obtained from σ_1' by filling in the boxes in σ_1' with monotypes. This judgement is called *boxy matching* and we discuss it further in Section 3.5.

[1] We have not explored what a logical, rather than syntax-directed, presentation of our type system would look like.

$$\boxed{\vdash \sigma_1' \leq \sigma_2'}$$

$$\frac{\begin{array}{c}\sigma_1' \neq \boxed{\sigma} \quad \overline{b}\#\mathit{ftv}(\sigma_1') \\ \vdash \sigma_1' \leq \rho_2'\end{array}}{\vdash \sigma_1' \leq \forall\overline{b}\,.\,\rho_2'}\ \textsc{skol} \qquad \frac{\vdash [\overline{a \mapsto \boxed{\sigma}}]\rho_1' \leq \rho_2'}{\vdash \forall\overline{a}\,.\,\rho_1' \leq \rho_2'}\ \textsc{spec}$$

$$\frac{}{\vdash \tau \leq \tau}\ \textsc{mono} \qquad \frac{\vdash \boxed{\sigma} \sim \sigma'}{\vdash \boxed{\sigma} \leq \sigma'}\ \textsc{sboxy}$$

$$\frac{}{\vdash \tau \leq \boxed{\tau}}\ \textsc{bmono} \qquad \frac{\vdash T\ \overline{\sigma}' \sim \sigma'}{\vdash T\ \overline{\sigma}' \leq \sigma'}\ \textsc{con}$$

$$\frac{\vdash \sigma_3' \sim \sigma_1' \quad \vdash \sigma_2' \leq \sigma_4'}{\vdash \sigma_1' \to \sigma_2' \leq \sigma_3' \to \sigma_4'}\ \textsc{f}1 \qquad \frac{\vdash \sigma_1' \to \sigma_2' \leq \boxed{\sigma_3} \to \boxed{\sigma_4}}{\vdash \sigma_1' \to \sigma_2' \leq \boxed{\sigma_3 \to \sigma_4}}\ \textsc{f}2$$

$$\boxed{\vdash \sigma_1' \sim \sigma_2'}$$

$$\frac{\vdash \sigma_2' \sim \sigma_1'}{\vdash \sigma_1' \sim \sigma_2'}\ \textsc{sym} \qquad \frac{}{\vdash \boxed{\tau} \sim \boxed{\tau}}\ \textsc{bbeq}$$

$$\frac{}{\vdash \tau \sim \boxed{\tau}}\ \textsc{meq}1 \qquad \frac{}{\vdash \tau \sim \tau}\ \textsc{meq}2$$

$$\frac{\vdash \sigma_1' \to \sigma_2' \sim \boxed{\sigma_1} \to \boxed{\sigma_2}}{\vdash \sigma_1' \to \sigma_2' \sim \boxed{\sigma_1 \to \sigma_2}}\ \textsc{aeq}1 \qquad \frac{\vdash \sigma_1' \sim \sigma_3' \quad \sigma_2' \sim \sigma_4'}{\vdash \sigma_1' \to \sigma_2' \sim \sigma_3' \to \sigma_4'}\ \textsc{aeq}2$$

$$\frac{\vdash T\ \overline{\sigma_1'} \sim T\ \overline{\boxed{\sigma_2}}}{\vdash T\ \overline{\sigma_1'} \sim \boxed{T\ \overline{\sigma_2}}}\ \textsc{ceq}1 \qquad \frac{\vdash \overline{\sigma_1'} \sim \overline{\sigma_2'}}{\vdash T\ \overline{\sigma_1'} \sim T\ \overline{\sigma_2'}}\ \textsc{ceq}2$$

$$\frac{\vdash \rho_1' \sim \boxed{\rho_2}}{\vdash \forall\overline{a}\,.\,\rho_1' \sim \boxed{\forall\overline{a}\,.\,\rho_2}}\ \textsc{seq}1 \qquad \frac{\vdash \rho_1' \sim \rho_2'}{\vdash \forall\overline{a}\,.\,\rho_1' \sim \forall\overline{a}\,.\,\rho_2'}\ \textsc{seq}2$$

Figure 3: Subsumption and boxy matching

The second rule for lambda abstractions, ABS2, deals with the pure inference case; that is, when there is a box around the entire result type. In this case we "push down the box" and try again, thereby dispatching to ABS1. Operationally, if the result type is a hole, then we make two new holes for argument and body types, and type check the function; then read out the results from the holes, form these results into a function type, and write it into the result hole. (Incidentally, although the box is filled in with $\boxed{\sigma_1 \to \sigma_2}$, these types can only be of the form $\boxed{\rho_1 \to \rho_2}$ in all valid typing derivations.)

Turning now to application, consider rule APP. It type checks the function with the result type $\boxed{\sigma} \to \rho'$. This type asks to *infer* the type of the function *argument*, while the *result* type is inferred or checked as specified by ρ'. Once we know the argument type, σ, we can use $\vdash^{poly}$ to type check the argument itself, so that information from the function's type is pushed into the argument, just as discussed in Section 2.1. However, unlike the all-or-nothing approach of local type inference, boxy types support a mixture of inference and checking. For example, we can type the following:

$$\vdash \mathtt{(\backslash x.\backslash g.\ (g\ True,\ g\ 3))\ 4} : (\forall a\,.\,a \to b) \to (b, b)$$

Using the APP rule, the function is typed as follows; note that the judgement *infers* the argument type while *checking* the result type:

$$\vdash \texttt{(\textbackslash x.\textbackslash g. (g True, g 3))} : \boxed{\texttt{Int}} \rightarrow (\forall a . a \rightarrow b) \rightarrow (b, b)$$

3.3 Type annotations and lexically scoped type variables

The rule LET-S allows the programmer to specify the type of the right-hand side of a let binding instead of inferring that type. This is why in LET-S there is no box around ρ when u is checked, whereas in LET-I there is. An unusual feature is that the quantified variables $\overline{a}$ of the user-specified type $\forall \overline{a} . \rho$ *scope over other type annotations inside* u as well as over ρ; this is specified in LET-S by extending the environment Γ with the type variables $\overline{a}$. For example, the type signature on the declaration of `y` below mentions the variable `a` that is bound by the type signature for `id`:

```
let id :: forall a. a -> a
    id = \x . let y :: a = x in y
```

Scoped type variables, such as `a`, are "rigid"—they may not unify with any other types[2].

The side condition $ftv(\forall \overline{a} . \rho) \subseteq dom(\Gamma)$ in LET-S ensures that the type annotation only mentions type variables that are in scope, while $\overline{a} \# ftv(\Gamma)$ ensures that the new type variables are fresh.

Occasionally, we will use the annotation form $(t\ \texttt{::}\ \sigma)$, which is syntactic sugar for `let` x `::` σ `=` t `in` x.

3.4 Subsumption

The variable rule, VAR, must check the compatibility between a vanilla type σ found in the environment and the boxy type ρ' that is the result of the judgement. This compatibility check, called *subsumption* and generalised to two boxy polytypes, is performed by the judgement $\vdash \sigma_1' \leq \sigma_2'$ shown in Figure 3. This judgment holds if a value of type σ_1' is acceptable in a context needing a σ_2'; that is, if σ_1' is at least as polymorphic as σ_2'. Most of the action in our type system is in the definition of subsumption, and we devote the rest of this section to it.

A key principle guiding the design of our subsumption relation is that if we know (a) the type of a polymorphic function `f`, and (b) the type of `f`'s instance at a call site, then we should be able to work out the types at which `f` is instantiated, even if they are polytypes. More precisely:

Principle 2 (Arbitrary instantiation) $\vdash \forall \overline{a} . \rho \leq [\overline{a \mapsto \sigma}]\rho$ *for any box-free polytypes* $\overline{\sigma}$.

For example, say that σ_{id} abbreviates the type $\forall a . a \rightarrow a$. It must be that $\vdash \sigma_{id} \leq \sigma_{id} \rightarrow \sigma_{id}$, by instantiating a to σ_{id}. Obeying Principle 2 means that the programmer can, if all else fails, specify the instantiation of a polymorphic function by specifying the type of the result of the instantiation.

Our definition of subsumption is based on that of the Hindley-Milner type system. There, polytypes are of the form $\forall \overline{a} . \tau$ and subsumption is specified by three rules. The first two rules cover instantiation: guess monotypes to instantiate top-level type variables of the left-hand type and make sure that the resulting monotypes are equal.

$$\frac{\vdash [\overline{a \mapsto \tau}]\tau_1 \leq \tau_2}{\vdash \forall \overline{a} . \tau_1 \leq \tau_2}\ \textsc{hm-spec} \qquad \frac{}{\vdash \tau \leq \tau}\ \textsc{hm-mono}$$

The final rule allows a polytype on the right-hand side by adding a rule to "skolemise" that type variable (see for example [15]):

$$\frac{\overline{a} \# ftv(\sigma) \quad \vdash \sigma \leq \tau}{\vdash \sigma \leq \forall \overline{a} . \tau}\ \textsc{hm-skol}$$

In our system, we generalise this subsumption relation to support impredicative and higher-rank polymorphism in several ways.

The key difference from HM-SPEC is that the instantiation rule, SPEC in Figure 3, *instantiates the type on the left with boxy polytypes*, rather than monotypes. Algorithmically, we are trying to infer what *polytypes* should be used to instantiate the type variables, so we allocate a new box for each type variable, and recursively call the subsumption judgement to fill the boxes in. Pleasingly, the very same mechanism that allows us to propagate type information in support of higher-rank polymorphism (Section 3.2) is the key to impredicativity[3].

The rule for skolemisation, SKOL, is the same as HM-SKOL except for the side condition $\sigma_1' \neq \boxed{\sigma}$, which prevents overlap with rule SBOXY. We will discuss SBOXY shortly.

The rule for monotypes, MONO, is the same as HM-MONO.

Our relation also includes a number of rules that have no counterpart in the Hindley-Milner system. These rules explain how to deal with boxes and with polytypes that occur within function types and as the arguments to type constructors.

The rule CON deals with data types T, such as lists. We do not perform subsumption on the type arguments; instead we require that the argument types have compatible boxes, using the same boxy-matching judgement ($\vdash \sigma_1' \sim \sigma_2'$) that we encountered in rule ABS1. We discuss boxy matching in Section 3.5, but the idea is that it performs no instantiation or skolemisation. Thus, $\not\vdash [\forall a . a \rightarrow a] \leq [\texttt{Int} \rightarrow \texttt{Int}]$. This is a design choice—allowing *covariant* subsumption for some type constructors would be compatible with our system, but not with GHC. In the GHC core language, based on System F, performing such an instantiation would require mapping a type application down the list, and that is hard to do for arbitrary data types. Furthermore, such a mapping operation carries a run-time cost; sometimes it may be eliminated by type erasure, but even that is not true in the dictionary-passing implementation of Haskell type classes. In short, we require *invariance* in the type arguments of data types.

We allow a little more flexibility for functions, as rule F1 shows: it is *covariant* in the result type, but *invariant* in the argument type. Result covariance allows us to check the application f `2 True`, where $f : \forall a . a \rightarrow (\forall b . b \rightarrow b)$, by instantiating the type of f to $\texttt{Int} \rightarrow \texttt{Bool} \rightarrow \texttt{Bool}$. This is also a design choice; covariance for function return types buys us some extra functionality with little cost at run-time. However, making return types invariant would not destroy the essential properties of the system. The reader might wonder why we do not use contravariance in the argument type, as previous work has done, a question that we discuss in Section 7. The companion rule F2 pushes down a box on the right, just as ABS2 did.

Now consider rule SBOXY. Principle 2 requires that, for example, $\vdash \forall a\ .\ \texttt{Int} \rightarrow a \leq \texttt{Int} \rightarrow (\forall b . b \rightarrow b)$. Applying SPEC, we have $\vdash \texttt{Int} \rightarrow \boxed{\forall b . b \rightarrow b} \leq \texttt{Int} \rightarrow (\forall b . b \rightarrow b)$. Now applying F1 we need that $\vdash \boxed{\forall b . b \rightarrow b} \leq \forall b . b \rightarrow b$, and that motivates rule SBOXY. It says that when the type on the left is a box, we should simply fill in the box with type σ' on the right, but, just as in ABS1, we use boxy matching to extract a vanilla type from σ'. Note that there is no corresponding rule that fills in a box on the right with a polytype on the left. Instead, a box on the right can

[2] We have also explored the alternative of "flexible" scoped type variables, that stand for an arbitrary *type* rather than a *variable*. With this extension, the system remains tractable but becomes slightly more complicated.

[3] Incidentally, by instantiating with boxy types, rule SPEC gives rise to a boxy type on the left of the $\leq$, which is why we must allow the judgement to have boxy types on both sides.

only contain a monotype that appears on the left (see rule BMONO). In Section 7 we discuss why this is the case.

3.5 Boxy matching

Boxy matching, $\vdash \sigma_1' \sim \sigma_2'$, ensures that two types have “compatible boxes”. Its rules are given in Figure 3.

The key rule is the innocuous-looking BBEQ. This rule says that *when a box meets a box, we must fill them both in with a monotype*. Alternatively, if one type or the other has structure outside a box, then that structure is “known” and can be used to fill in the other box; that is what is happening in rules MEQ1, AEQ1, CEQ1, and SEQ1. These rules together may derive $\vdash \sigma \sim \boxed{\sigma}$ for any polytype. The rule SYM ensures that MEQ1, AEQ1 etc. also apply when the box is on the left. If both types have known structure outside the box, then matters are even more straightforward (rules MEQ2, AEQ2, CEQ2, and SEQ2.) In rule CEQ2, the notation $\vdash \overline{\sigma_1'} \sim \overline{\sigma_2'}$ is the pointwise application of boxy matching to the elements of $\overline{\sigma_1'}$ and $\overline{\sigma_2'}$. If on the other hand neither type has known structure then BBEQ must simply “guess”, and simple inference can only guess a monotype.

The “box meets box” situation implies $\not\vdash a \to \boxed{\sigma_{id}} \sim \boxed{a \to \sigma_{id}}$, because otherwise we need $\vdash \boxed{\sigma_{id}} \sim \boxed{\sigma_{id}}$. On the other hand, $\vdash \sigma_{id} \to \boxed{\sigma_{id}} \sim \boxed{\sigma_{id}} \to \sigma_{id}$ because the boxes are matched up against box-free types.

Algorithmically, boxy matching can be thought of as a “super-unifier” for boxy types: the two structures are unified, with the known structure of each filling the holes in the other.

4. Properties of the type system

Having presented the basic syntax and inference rules, we now elaborate on the properties of this type system. All the results we mention are presented with detailed proofs in an accompanying technical report [26].

4.1 Reflexivity and transitivity

By design, boxy matching is not an equivalence relation. It is not reflexive, for example $\not\vdash \boxed{\sigma_{id}} \sim \boxed{\sigma_{id}}$ because this would require guessing polytype information. Likewise, it is not transitive: $\vdash \boxed{\sigma_{id}} \sim \sigma_{id}$ and $\vdash \sigma_{id} \sim \boxed{\sigma_{id}}$ but $\not\vdash \boxed{\sigma_{id}} \sim \boxed{\sigma_{id}}$. For the same reason, subsumption is not reflexive or transitive. For example, $\not\vdash \boxed{\sigma_{id}} \leq \boxed{\sigma_{id}}$. Also, $\vdash \boxed{\sigma_{id}} \to \mathtt{Int} \leq \sigma_{id} \to \mathtt{Int}$ and $\vdash \sigma_{id} \to \mathtt{Int} \leq \boxed{\sigma_{id}} \to \mathtt{Int}$ but $\not\vdash \boxed{\sigma_{id}} \to \mathtt{Int} \leq \boxed{\sigma_{id}} \to \mathtt{Int}$. For box-free types, both matching and subsumption are reflexive because no guessing is required. Likewise, boxy matching is also transitive for box-free types.

However, subsumption is *not* transitive for box-free types because the SPEC rule introduces boxes. For example, $\vdash \sigma_{id} \leq \sigma_{id} \to \sigma_{id}$ and $\vdash \sigma_{id} \to \sigma_{id} \leq \sigma_{id} \to \sigma_{id} \to \sigma_{id}$ but $\not\vdash \sigma_{id} \leq \sigma_{id} \to \sigma_{id} \to \sigma_{id}$. Intuitively, once a box has been filled in with a type, that type cannot be further instantiated.

This restriction implies, for example, that we can check that the id function has type σ_{id}, and $\sigma_{id} \to \sigma_{id}$, but not $\sigma_{id} \to \sigma_{id} \to \sigma_{id}$. In general it is *not true* that:

$$\Gamma \vdash t : \rho_1' \text{ and } \vdash \rho_1' \leq \rho_2' \text{ implies } \Gamma \vdash t : \rho_2'$$

However, the loss of transitivity has no impact on type inference. The completeness of our algorithm only requires a restriction of box-free transitivity, a point we discuss further in the technical report [26].

4.2 Dropping and expanding boxes

Providing additional (correct) type information should not make programs untypeable, cause subsumption not to hold, nor make types fail to match. Technically, one way to provide more information is to remove boxes from types in judgements. For example, if the derivation $\vdash \sigma \to \sigma \leq \boxed{\sigma} \to \sigma$ holds, we should be able to derive $\vdash \sigma \to \sigma \leq \sigma \to \sigma$. Another way to make more type information checkable is to push a box down the syntax of a type. For example, if $\vdash \boxed{\sigma \to \sigma} \leq \sigma \to \sigma$, it should also be the case that $\vdash \boxed{\sigma} \to \boxed{\sigma} \leq \sigma \to \sigma$. In general, if inference succeeds with partial type information, it should succeed when some of that inferred type information becomes checkable.

To formalise this property, we define the following relation between types. We say that σ_2' is the *unboxing* of σ_1', written $\vdash \sigma_1' \blacktriangleright \sigma_2'$, if σ_2' can be constructed from σ_1' by eliminating boxes or pushing the boxes down the abstract syntax of the type.[4] For example, $\vdash \boxed{a} \blacktriangleright a$ and $\vdash \boxed{\sigma_{id} \to \sigma_{id}} \blacktriangleright \sigma_{id} \to \boxed{\sigma_{id}}$.

We can then prove that unboxing types preserves boxy matching, subsumption, and typability.

Lemma 4.1 (Unboxing)

1. *If* $\vdash \sigma_1' \sim \sigma_2'$ *and* $\vdash \sigma_1' \blacktriangleright \sigma_3'$ *and* $\vdash \sigma_2' \blacktriangleright \sigma_4'$ *then* $\vdash \sigma_3' \sim \sigma_4'$.
2. *If* $\vdash \sigma_1' \leq \sigma_2'$ *and* $\vdash \sigma_1' \blacktriangleright \sigma_3'$ *and* $\vdash \sigma_2' \blacktriangleright \sigma_4'$ *then* $\vdash \sigma_3' \leq \sigma_4'$.
3. *If* $\Gamma \vdash t : \rho_1'$ *and* $\vdash \rho_1' \blacktriangleright \rho_2'$ *then* $\Gamma \vdash t : \rho_2'$.

Conversely, sometimes it is acceptable to box parts of types or expand boxes up the tree. For example, the judgement $\vdash \sigma_{id} \leq \mathtt{Int} \to \mathtt{Int}$ holds, and so does $\vdash \sigma_{id} \leq \boxed{\mathtt{Int}} \to \boxed{\mathtt{Int}}$ and $\sigma_{id} \leq \boxed{\mathtt{Int} \to \mathtt{Int}}$. However, boxing and box expansion is only valid for monotype information. Recall that boxes in types represent “guessed” information, and only monotypes may be arbitrarily guessed. For example, even though $\vdash \sigma_{id} \leq \sigma_{id} \to \sigma_{id}$, it is not the case that $\vdash \sigma_{id} \leq \boxed{\sigma_{id}} \to \boxed{\sigma_{id}}$ or $\vdash \sigma_{id} \leq \boxed{\sigma_{id} \to \sigma_{id}}$.

Consequently, we can add or expand boxes—as long as we do not put a box around a polytype that was not originally boxed. In this case, we define σ_2' to be the *monotype boxing* of σ_1', written $\vdash \sigma_1' \triangleleft \sigma_2'$, when $\vdash \sigma_2' \blacktriangleright \sigma_1'$ and the quantifiers of σ_2' appear inside a box only if the corresponding quantifiers in σ_1' do. For example, $\vdash a \triangleleft \boxed{a}$ but $\not\vdash \sigma_{id} \to \boxed{\sigma_{id}} \triangleleft \boxed{\sigma_{id} \to \sigma_{id}}$. Like unboxing, monotype boxing preserves boxy matching, subsumption and typing.

Lemma 4.2 (Monotype Boxing)

1. *If* $\vdash \sigma_1' \sim \sigma_2'$ *and* $\vdash \sigma_1' \triangleleft \sigma_3'$ *and* $\vdash \sigma_2' \triangleleft \sigma_4'$ *then* $\vdash \sigma_3' \sim \sigma_4'$.
2. *If* $\vdash \sigma_1' \leq \sigma_2'$ *and* $\vdash \sigma_1' \triangleleft \sigma_3'$ *and* $\vdash \sigma_2' \triangleleft \sigma_4'$ *then* $\vdash \sigma_3' \leq \sigma_4'$.
3. *If* $\Gamma \vdash t : \rho_1'$ *and* $\vdash \rho_1' \triangleleft \rho_2'$ *then* $\Gamma \vdash t : \rho_2'$.

Lemmas 4.1 and 4.2 confirm the intuition that *boxes do not matter for monotype information*. Monotypes may be arbitrarily boxed or unboxed without destroying the validity of a judgment.

4.3 Embedding of System F

We began by taking System F as our “gold standard”, so it is natural to ask whether every System F program can be expressed in our language. Happily, this is the case, provided that the programmer adds sufficient type annotations. The guideline for programmers to write any System F program is that it is enough to put annotations (a) on functions that take polymorphic arguments, (b) on impredicative type instantiations, and (c) on type lambdas that bind type variables that appear in other annotations. Of course, in many situations, we expect that fewer annotations will be necessary.

Figure 4 shows the type-directed embedding of System F terms. In the figure, note that the translation of type abstractions often requires a type annotation. The reason is that the bound type variable

[4] The formal definition of this relation can be found in the technical report.

$$\boxed{[\![t^F]\!]_{\Gamma^F} = t}$$

$$\begin{array}{lcll}
[\![x]\!]_\Gamma & = & x & \\
[\![\lambda x\,.\,t]\!]_\Gamma & = & \backslash x\,.\,[\![t]\!]_{\Gamma,x:\tau} & \text{where } \Gamma, x{:}\tau \vdash^F t : \sigma_2 \\
[\![\lambda x\,.\,t]\!]_\Gamma & = & (\backslash x\,.\,[\![t]\!]_{\Gamma,x:\sigma_1})::\sigma_1 \rightarrow \sigma_2 & \\
 & & & \text{where } \Gamma, x{:}\sigma_1 \vdash^F t : \sigma_2 \\
[\![t_1\ t_2]\!]_\Gamma & = & [\![t_1]\!]_\Gamma\ [\![t_2]\!]_\Gamma & \\
[\![\Lambda a\,.\,t]\!]_\Gamma & = & [\![t]\!]_{\Gamma,a} & \text{where } a \notin ftv([\![t]\!]_{\Gamma,a}) \\
[\![\Lambda a\,.\,t]\!]_\Gamma & = & [\![t]\!]_{\Gamma,a}::\sigma & \text{where } \Gamma \vdash^F \Lambda a\,.\,t : \sigma \\
[\![t\ \tau]\!]_\Gamma & = & [\![t]\!]_\Gamma & \\
[\![t\ \sigma]\!]_\Gamma & = & [\![t]\!]_\Gamma::\sigma_1 & \text{where } \Gamma \vdash^F t\ \sigma : \sigma_1
\end{array}$$

Figure 4: Embedding of System F

may appear inside the *translated* body, if for example it is part of a higher-rank annotation. The annotation merely brings that variable in scope[5]. The typing relation of System F ($\Gamma \vdash^F t : \sigma$) is standard and we omit it for reasons of space.

Theorem 4.3 (Embedding of System F) *If* $\Gamma \vdash^F t : \sigma$ *then* $\Gamma \vdash^{poly} [\![t]\!]_\Gamma : \sigma$ *and* $\Gamma \vdash^{poly} [\![t]\!]_\Gamma : \boxed{\rho}$ *where* $prenex(\sigma) = \forall \overline{a}\,.\,\rho$.

The first part of the theorem states that we can *check* that the translated term indeed is typeable with the System F type. The second part states that we can we can *infer* an equivalent type.[6]

4.4 Type Soundness

We define the dynamic semantics of this language through a straightforward translation to System F, written $\Gamma \vdash t : \sigma' \rightsquigarrow t'$. This translation—omitted due to lack of space—is essentially the identity plus type coercions whose behavior is that of the identity function. These coercion functions are induced by the translation of the subsumption judgement, $\vdash \sigma_1' \leq \sigma_2' \rightsquigarrow t$. Therefore, we prove the type soundness of our language with respect to this semantics by showing that this translation is type-preserving. Below, the function $strip(\cdot)$ merely removes the boxes from a type.

Lemma 4.4 (Term translation)

1. *If* $\vdash \sigma_1' \leq \sigma_2' \rightsquigarrow t$ *then* $\vdash^F t : strip(\sigma_1' \rightarrow \sigma_2')$.
2. *If* $\Gamma \vdash t : \rho' \rightsquigarrow t'$ *then* $\Gamma \vdash^F t' : strip(\rho')$.
3. *If* $\Gamma \vdash^{poly} t : \sigma' \rightsquigarrow t'$ *then* $\Gamma \vdash^F t' : strip(\sigma')$.

Defining the dynamic semantics through translation is convenient for two reasons. First, this translation semantics is the semantics that GHC actually implements—we would have to show that any other semantics is equivalent to this one to argue that GHC actually implements this language. Indeed, our translation demonstrates that our type system is suitable for compilers, such as GHC, that use intermediate languages based on System F. Second, this translation simplifies the proof that our type system is type sound. The standard way to show type safety with a direct semantics is to show that it satisfies the properties of *subject reduction* and *progress*. Unsurprisingly for a language that relies on type annotations, subject reduction fails with a straightforward substitution-based semantics. For example, take $\sigma = (\forall a\,.\,a \rightarrow a) \rightarrow (\texttt{Int}, \texttt{Bool})$. We have

$$\vdash \texttt{let}\ f::\sigma = (\backslash g\,.\,(g\ 3, g\ \texttt{True}))\ \texttt{in}\ f\ (\backslash x\,.\,x) : (\texttt{Int}, \texttt{Bool})$$

but $\quad \not\vdash (\backslash g\,.\,(g\ 3, g\ \texttt{True}))\ (\backslash x\,.\,x) : (\texttt{Int}, \texttt{Bool})$
since the annotation required to give $\backslash g\,.\,(g\ 3, g\ \texttt{True})$ a higher-rank type is no longer present. Instead, to prove type soundness, we would have to cleverly devise a reduction relation that preserves type annotations. Such a semantics would be useful to explain to programmers how to transform their code while retaining typability. However, we leave that as future work (see Section 9) and stick with the simpler, transformation-based semantics for now.

4.5 Extension of Hindley-Milner

We have proved that the type system presented in Section 3 is a conservative extension of Hindley-Milner. In particular, if an expression type checks in Hindley-Milner with no user annotations, then our type system will infer its type. Conversely, derivations in the subset of our language that does not use annotations or higher-rank types in environments correspond to derivations in the Hindley-Milner system. The proofs of these theorems crucially rely upon the boxing and unboxing lemmas (4.2 and 4.1).

Theorem 4.5 (Conservative extension of HM) *Assume that* Γ *contains no higher-rank types and* t *contains no type annotations. Then* $\Gamma \vdash^{HM} t : \tau$ *iff* $\Gamma \vdash t : \tau$.

5. Type inference

The type system that we presented in Section 3 has an inference algorithm that is a modest elaboration of the classic Damas-Milner Algorithm $\mathcal{W}$ [2]. The details of the algorithm can be found in the technical report [26].

The algorithm distinguishes between ordinary unification variables, denoted with α, β, which can range only over monotypes, and boxy unification variables, denoted with ζ, ξ which can range over arbitrary types. Boxes appearing in the specification correspond to boxy unification variables in the algorithm. Ordinary unification variables force boxes to contain (perhaps unknown yet) monotypes. Unifiers, denoted with S, are idempotent substitutions from unification variables (boxy or not) to box-free types. Additionally, the algorithm makes use of an infinite sequence of symbols, denoted $\mathcal{A}$, for unification variables or skolem constants.

The main algorithm is presented as a deterministic relation:

$$(S_0, \mathcal{A}_0) \succ \Gamma \vdash t : \rho' \succ (S_1, \mathcal{A}_1)$$

The judgement should be read as: "given an initial unifier S_0 and symbol supply $\mathcal{A}_0$, check that t has the type ρ' under Γ, returning an extended unifier S_1 and remaining symbol supply $\mathcal{A}_1$". Boxy matching and subsumption can also be presented as deterministic relations:

$$(S_0, \mathcal{A}_0) \succ \vdash \sigma_1' \leq \sigma_2' \succ (S_1, \mathcal{A}_1)$$
$$(S_0, \mathcal{A}_0) \succ \vdash \sigma_1' \sim \sigma_2' \succ (S_1, \mathcal{A}_1)$$

The syntax of types ρ' and σ' appearing in algorithm judgements is different than the syntax of the specification—namely they are allowed to contain ordinary and boxy unification variables, but no boxes. Type variables appearing in such types correspond to skolem constants. The free type variables (*ftv*) of such a type include both unification and normal type variables.

A crucial operational invariant is this: *every invocation of the inference algorithm fills in all the holes (i.e. boxy unification variables) in its input*. We think of the holes as the out-parameters of the algorithm. More precisely:

Theorem 5.1 *If* $(S_0, \mathcal{A}_0) \succ \Gamma \vdash t : \rho' \succ (S_1, \mathcal{A}_1)$ *then the free boxy unification variables of* ρ' *are in* $dom(S_1)$.

[5] With "existential" annotations [20, 21], we would not need to annotate type abstractions.

[6] The prenex form of a type is one where all the quantifiers on the right hand sides of arrows have been pulled to the top in a capture-avoiding way.

It follows that, after the invocation of the algorithm, we can safely read-off the boxy variables from the returned unifier. As an example, consider the algorithm rule for `let`:

$$\frac{\begin{array}{c}(S_0, \mathcal{A}_0) \succ \Gamma \vdash u : \zeta \succ (S_1, \mathcal{A}_1) \\ X = \mathit{ftv}(S_1\zeta) - \mathit{ftv}(S_1\Gamma) \\ (S_1, \mathcal{A}_1) \succ \Gamma, x{:}\forall\overline{a}\,.\,[\overline{X \mapsto a}]S_1\zeta \vdash t : \rho' \succ (S_2, \mathcal{A}_2)\end{array}}{(S_0, \mathcal{A}_0\overline{a}\zeta) \succ \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho' \succ (S_2, \mathcal{A}_2)}\ \textsc{ALET-I}$$

First, a new boxy variable ζ is extracted from the top of the initial supply $\mathcal{A}_0\overline{a}\zeta$, and u is type checked with the initial unifier. Then we can safely read off the value $S_1\zeta$, generalise this type over $S_1\Gamma$, and check the body of the expression. Note that we do not in general apply the substitution to the environment, but rather thread it lazily through our judgements.

We have proven that the algorithm is sound and complete with respect to the specification given in Section 3. Soundness asserts that if the algorithm succeeds, then there exists a corresponding derivation in the specification.

Theorem 5.2 (Soundness) *If $\mathcal{A}_0$ is a fresh symbol supply and $(\emptyset, \mathcal{A}_0) \succ \vdash t : \rho' \succ (S, \mathcal{A}_1)$ then $\vdash t : [\![\rho']\!]_S$.*

The operation $[\![\cdot]\!]_S$ applies the substitution S to an algorithmic type (which can contain unification variables, but not boxes) to obtain a specification type (which has boxes instead of boxy unification variables). Completeness asserts that if there is a derivation in the specification, then the algorithm succeeds and returns a most general unifier.

Theorem 5.3 (Completeness) *If $\vdash\ t\ :\ [\![\rho']\!]_S$ and $\mathcal{A}_0$ is a fresh symbol supply then $(\emptyset, \mathcal{A}_0)\ \succ\ \vdash t : \rho'\ \succ\ (S_0, \mathcal{A}_1)$ such that $\exists R\,.\,S = (R \cdot S_0)\backslash_{\mathcal{A}_0 - \mathcal{A}_1}$.*

The notation $S\ =\ (R \cdot S_0)\backslash_{\mathcal{A}_0 - \mathcal{A}_1}$ states that S and the composition of R and S_0 may disagree *only* on meta variables coming from the set $\mathcal{A}_0 - \mathcal{A}_1$. Theorem 5.3 implies that in pure inference mode, that is, when $[\![\rho']\!]_S$ is just a box, $\boxed{\rho}$, we can simply initialise the algorithm with a fresh boxy variable ζ.

Soundness and completeness of the algorithm give us a principal types property for our specification. The theorem below states that *for a given amount of checked information* there exists a "best" amount of inferred information for a given term.

Theorem 5.4 (Principal Types) *Suppose that $\vdash t : \rho'$. Then there exists a ρ'_0 such that $\vdash t : \rho'_0$, and for all ρ'_1 that differ from ρ' only inside boxes and $\vdash\ t\ :\ \rho'_1$ it is the case that $\rho'_1\ =\ R\rho'_0$ for some substitution R.*

6. Reducing type annotations

The type system of Section 3 is sufficient to type all programs involving impredicative and higher-rank polymorphism, but the type annotations can sometimes be tedious. In this section we describe some extensions to the type system that help eliminate some of these annotations. Except where noted, all of these extensions satisfy the properties discussed in Sections 4 and 5.

For example, consider the expression `tail ids`, where $\texttt{ids} : \texttt{[}\sigma_{id}\texttt{]}$, and $\texttt{tail} : \forall b\,.\,\texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]}$ (recall that $\sigma_{id} = \forall a\,.\,a \to a$). Even if we know the result type, we cannot produce a derivation:

$$\frac{\dfrac{\nvdash \forall b\,.\,\texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]} \leq \boxed{\texttt{[}\sigma_{id}\texttt{]}} \to \texttt{[}\sigma_{id}\texttt{]}}{\Gamma \vdash \texttt{tail} : \boxed{\texttt{[}\sigma_{id}\texttt{]}} \to \texttt{[}\sigma_{id}\texttt{]}}\ \textsc{VAR} \qquad \Gamma \vdash^{poly} \texttt{ids} : \texttt{[}\sigma_{id}\texttt{]}}{\Gamma \vdash \texttt{tail}\ \texttt{ids} : \texttt{[}\sigma_{id}\texttt{]}}\ \textsc{APP}$$

We run into trouble at the VAR rule because no derivation exists for the instantiation of `tail`. Rule SPEC cannot guess a polymorphic instantiation in the judgment

$$\vdash \forall b\,.\,\texttt{[}b\texttt{]} \to \texttt{[}b\texttt{]} \leq \boxed{\texttt{[}\sigma_{id}\texttt{]}} \to \texttt{[}\sigma_{id}\texttt{]}$$

because that requires the derivation $\vdash\ \texttt{[}\boxed{\sigma_{id}}\texttt{]}\ \sim\ \boxed{\texttt{[}\sigma_{id}\texttt{]}}$. The only way to make this example type check is to add an explicit annotation to `tail`, forcing the impredicative instantiation:

$$\Gamma \vdash (\texttt{tail} :: \texttt{[}\sigma_{id}\texttt{]} \to \texttt{[}\sigma_{id}\texttt{]})\ \texttt{id} : \texttt{[}\sigma_{id}\texttt{]}$$

The trouble arises because APP and VAR are done separately. If we combine them into a single rule, all is well:

$$\frac{\begin{array}{c}\nu{:}\forall\overline{a}\,.\,\overline{\sigma} \to \sigma \in \Gamma \\ \Gamma \vdash^{poly} u_i : [\overline{a \mapsto \boxed{\sigma}}]\sigma_i \qquad \vdash [\overline{a \mapsto \boxed{\sigma}}]\sigma \leq \rho'\end{array}}{\Gamma \vdash \nu\ \overline{u} : \rho'}\ \textsc{SMART-APP}$$

Notice that VAR is just a special case of SMART-APP with zero arguments. The type system of Figure 2 could be recast by completely dropping VAR and replacing it with rule SMART-APP.

6.1 Exploiting flow in application nodes

We can do better! SMART-APP fails for $\Gamma \vdash \texttt{sing}\ \texttt{id} : \texttt{[}\sigma_{id}\texttt{]}$ where $\Gamma\ =\ \texttt{sing}\ :\ \forall b\,.\,b\ \to\ \texttt{[}b\texttt{]}, \texttt{id}\ :\ \sigma_{id}$. The reason is that type checking the argument requires $\Gamma \vdash \texttt{id} : \boxed{\sigma_{id}}$, and that fails.

This failure is frustrating, because once the *result* type of the call to `sing id` is fixed, we know its instantiation. That suggests the following variant of SMART-APP—use knowledge about the result type of the call to fix at least part of the instantiation of the function:

$$\frac{\begin{array}{c}\nu{:}\forall\overline{a}\,.\,\overline{\sigma} \to \sigma \in \Gamma \\ \overline{a}_c = \overline{a} \cap \mathit{ftv}(\sigma) \quad \overline{a}_{rest} = \overline{a} - \overline{a}_c \\ \vdash [\overline{a_c \mapsto \boxed{\sigma_c}}]\sigma \leq \rho' \\ \Gamma \vdash^{poly} u_i : [\overline{a_{rest} \mapsto \boxed{\sigma_r}}, \overline{a_c \mapsto \sigma_c}]\sigma_i\end{array}}{\Gamma \vdash \nu\ \overline{u} : \rho'}\ \textsc{SA-RES}$$

First we find $\overline{a}_c$, the type variables that are mentioned in the result type of the function, σ. Then, we use the subsumption judgement to compare σ with the context type ρ', to produce the instantiation types $\overline{\sigma_c}$. These types can then be used, *sans-box*, to instantiate the argument types, while the remaining instantiations $\overline{\sigma_r}$ remain boxed. In this way we can take advantage of shape information in the result type, and push that information into the arguments. In particular, SA-RES can type `sing id`:

$$\frac{\vdash \texttt{[}\boxed{\sigma_{id}}\texttt{]} \leq \texttt{[}\sigma_{id}\texttt{]} \quad \Gamma \vdash^{poly} \texttt{id} : \sigma_{id}}{\Gamma \vdash \texttt{sing}\ \texttt{id} : \texttt{[}\sigma_{id}\texttt{]}}\ \textsc{SA-RES}$$

Rule SA-RES is useful, but not entirely satisfactory. For example it fails to check $\Gamma \vdash^{poly} \texttt{head}\ \texttt{ids} : \forall a\,.\,a \to a$. In general, because rule GEN performs skolemisation (here for the type $\forall a\,.\,a\ \to\ a$), the system fails to check applications where ν has a type of the form $\forall\overline{a}\,.\,\overline{\sigma} \to a$, and a is instantiated with a polytype.

To solve this problem, we could work the other way around, so that we use the *argument types* to instantiate the function instead of the *result type*, thus:

$$\frac{\begin{array}{c}\nu{:}\forall\overline{a}\,.\,\overline{\sigma} \to \sigma \in \Gamma \\ \overline{a}_{arg} = \overline{a} \cap \mathit{ftv}(\overline{\sigma}) \quad \overline{a}_{rest} = \overline{a} - \overline{a}_{arg} \\ \Gamma \vdash^{poly} u_i : [\overline{a_{arg} \mapsto \boxed{\sigma_a}}]\sigma_i \\ \vdash [\overline{a_{arg} \mapsto \sigma_a}, \overline{a_{rest} \mapsto \boxed{\sigma_r}}]\sigma \leq \rho'\end{array}}{\Gamma \vdash \nu\ \overline{u} : \rho'}\ \textsc{SA-ARG}$$

This variant is incomparable to SA-RES. For example, SA-ARG can type $\Gamma \vdash^{poly} \texttt{head}\ \texttt{ids} : \sigma_{id}$, which SA-RES cannot; and SA-RES can type $\Gamma \vdash \texttt{cons}\ (\backslash x\,.\,x)\ \texttt{ids} : \texttt{[}\sigma_{id}\texttt{]}$, that SA-ARG cannot.

6.2 Unbiased smart application

Both SA-RES and SA-ARG fail for some relatively simple programs, so it is not clear which rule to use. In this section we show how to combine their virtues. The idea is this: in a typing judgement of the form $\Gamma \vdash \nu\ \overline{u} : \rho'$ we may be able to draw information about the instantiation of ν's type from the context type ρ' *and* the types of the arguments.

This indicates our approach: As a first step, a simple matching procedure, which we call *pre-subsumption*, draws information from the context type to be used in checking the arguments. Subsequently, the information gained from checking the arguments fills in boxes back in the context type.

However, during the first matching step, we must be careful to match type variables exclusively with the *invariant* parts of the context type. The reason is that covariant parts are subject to further instantiation, or skolemisation, as the following example indicates:

$$\Gamma \vdash ((\texttt{head ids})\texttt{::}\sigma_{id}) : \texttt{Int} \to \texttt{Int}$$

The type of `head` is $\forall a\,.\,[a] \to a$ and we would have to match the variable a with the skolemised type of σ_{id}, $b \to b$, which would lead to the wrong instantiation of a to $b \to b$, instead of the desired instantiation of a to σ_{id}.

Concretely, the rule is the following:

$$\dfrac{\begin{array}{cc} \nu{:}\forall\overline{a}\,.\,\overline{\sigma} \to \sigma \in \Gamma & \overline{a} \vdash \sigma \leq \rho' \Rightarrow \psi_0 \\ \psi = \psi_0 \sqcup \overline{[a \mapsto \boxed{\sigma}]} & \Gamma \vdash^{poly} u_i : \psi(\sigma_i) \\ \multicolumn{2}{c}{\psi_u = [a_{arg} \mapsto \mathit{strip}(\psi(a_{arg}))] \sqcup \psi} \\ \overline{a}_{arg} = \overline{a} \cap \mathit{ftv}(\overline{\sigma}) & \vdash \psi_u(\sigma) \leq \rho' \end{array}}{\Gamma \vdash \nu\ \overline{u} : \rho'}\ \text{SA-UNB}$$

In this rule, we first try to match the variables of $\overline{a}$ that appear in the result type σ with invariant parts of ρ' using pre-subsumption $\overline{a} \vdash \sigma \leq \rho' \Rightarrow \psi_0$. This judgement produces a substitution ψ_0 than maps some of the variables in $\overline{a}$ to boxy types. The details of pre-subsumption are in Figure 5, which we discuss shortly. Next, we create a substitution ψ for the entire $\overline{a}$ using ψ_0 plus guesses for any variables about which ψ_0 is silent. (For example, some $\overline{a}$ may not appear in the return type σ, or, even if they do, pre-subsumption may provide no information about them). Next, we check each argument u_i against the appropriate type $\psi(\sigma_i)$. These checks fill the boxes in ψ that correspond to variables that appear in the argument types $\overline{\sigma}$. Next, we create a new substitution, ψ_u, that propagates information from the arguments to the result type. This substitution is the same as ψ, except for the variables $\overline{a}_{arg}$. Because these variables appear in the argument types we know that their boxes were filled in. So we can "read-off" those types and add them to ψ_u sans-box. Finally, we perform the subsumption check $\vdash \psi_u(\sigma) \leq \rho'$.

It remains to explain how the pre-subsumption of Figure 5 works. The intuition behind the judgement $\overline{a} \vdash \sigma_1' \leq \sigma_2' \Rightarrow \psi$ is that ψ is a substitution for $\overline{a} \in \mathit{ftv}(\sigma_1')$ that maps them to "parts" of σ_2' that contain *known* polymorphic information—that is, polymorphic information outside boxes. For example,

$$a \vdash (a, a) \leq (\sigma_{id} \to \boxed{\sigma_{id}}, \boxed{\sigma_{id}} \to \sigma_{id}) \Rightarrow [a \mapsto (\sigma_{id} \to \sigma_{id})]$$

Pre-subsumption is straightforward—it traverses the structure of types, in the same manner as the subsumption judgment, identifying the invariant parts and in those places deferring to the *pre-matching* judgment (also in Figure 5) to create the substitution. Multiple substitutions are joined together with the operation $\psi_1 \sqcup \psi_2$ that creates a new substitution containing as much information as possible.

Pre-substitution and pre-matching are nondeterministic operations and many different substitutions may result for a given input.

$$\boxed{\text{Least boxy type } \sigma_1' \sqcup \sigma_2' = \sigma'}$$

$$\dfrac{}{\boxed{\sigma_1} \sqcup \sigma_2' = \sigma_2'}\ \text{LBT1} \qquad \dfrac{}{\sigma_1' \sqcup \boxed{\sigma_2} = \sigma_1'}\ \text{LBT2}$$

$$\dfrac{}{\sigma_1 \sqcup \sigma_2 = \sigma_2}\ \text{LBT3}$$

$$\dfrac{\sigma_1' \sqcup \sigma_3' = \sigma_{13}' \quad \sigma_2' \sqcup \sigma_4' = \sigma_{24}'}{\sigma_1' \to \sigma_2' \sqcup \sigma_3' \to \sigma_4' = \sigma_{13}' \to \sigma_{24}'}\ \text{LBT4}$$

$$\dfrac{\rho_1' \sqcup \rho_2' = \rho'}{\forall\overline{a}\,.\,\rho_1' \sqcup \forall\overline{a}\,.\,\rho_2' = \forall\overline{a}\,.\,\rho'}\ \text{LBT5}$$

$$\boxed{\text{Least boxy substitution } \psi_1 \sqcup \psi_1}$$

$$\psi = \psi_1 \sqcup \psi_2 \text{ when}$$

$$\begin{array}{ll} \psi(a) = \psi_1(a) & a \notin \mathit{dom}(\psi_2) \\ \psi(a) = \psi_2(a) & a \notin \mathit{dom}(\psi_1) \\ \psi(a) = \psi_1(a) \sqcup \psi_2(a) & a \in \mathit{dom}(\psi_1) \wedge a \in \mathit{dom}(\psi_2) \end{array}$$

$$\boxed{\text{Pre-matching } \overline{a} \vdash \sigma_1' = \sigma_2' \Rightarrow \psi_1}$$

$$\dfrac{}{\overline{a} \vdash \sigma_1' = \sigma_2' \Rightarrow \cdot}\ \text{M-ANY} \qquad \dfrac{a \in \overline{a}}{\overline{a} \vdash a = \sigma' \Rightarrow [a \mapsto \sigma']}\ \text{M-VAR}$$

$$\dfrac{\overline{a} \vdash \sigma_1' = \sigma_3' \Rightarrow \psi_1 \quad \overline{a} \vdash \sigma_2' = \sigma_4' \Rightarrow \psi_2}{\overline{a} \vdash \sigma_1' \to \sigma_2' = \sigma_3' \to \sigma_4' \Rightarrow (\psi_1 \sqcup \psi_2)}\ \text{M-FUN}$$

$$\dfrac{\overline{a} \vdash \sigma_{1i}' = \sigma_{2i}' \Rightarrow \psi_i \quad i = 1..n}{\overline{a} \vdash T\ \overline{\sigma}_1'^{1..n} = T\ \overline{\sigma}_2'^{1..n} \Rightarrow (\sqcup_{1..n}\psi_i)}\ \text{M-CON}$$

$$\dfrac{\overline{a} \vdash \rho_1' = \rho_2' \Rightarrow \psi \quad \overline{b}\ \#\ (\overline{a} \cup \mathit{range}(\psi))}{\overline{a} \vdash \forall\overline{b}\,.\,\rho_1' = \forall\overline{b}\,.\,\rho_2' \Rightarrow \psi}\ \text{M-ALL}$$

$$\boxed{\text{Pre-subsumption } \overline{a} \vdash \sigma_1' \leq \sigma_2' \Rightarrow \psi}$$

$$\dfrac{}{\overline{a} \vdash \sigma_1' \leq \sigma_2' \Rightarrow \cdot}\ \text{S-ANY}$$

$$\dfrac{\begin{array}{c} \sigma_1' \neq \boxed{\sigma} \\ \overline{a} \vdash \sigma_1' \leq \rho_2' \Rightarrow \psi \quad \overline{b}\#(\overline{a} \cup \mathit{range}(\psi)) \end{array}}{\overline{a} \vdash \sigma_1' \leq \forall\overline{b}\,.\,\rho_2' \Rightarrow \psi}\ \text{S-SKOL}$$

$$\dfrac{\overline{a} \vdash \overline{[b \mapsto \boxed{\sigma}]}\rho_1' \leq \rho_2' \Rightarrow \psi}{\overline{a} \vdash \forall\overline{b}\,.\,\rho_1' \leq \rho_2' \Rightarrow \psi}\ \text{S-SPEC}$$

$$\dfrac{\overline{a} \vdash \sigma_1' = \sigma_3' \Rightarrow \psi_1 \quad \overline{a} \vdash \sigma_2' \leq \sigma_4' \Rightarrow \psi_2}{\overline{a} \vdash \sigma_1' \to \sigma_2' \leq \sigma_3' \to \sigma_4' \Rightarrow (\psi_1 \sqcup \psi_2)}\ \text{S-FUN}$$

$$\dfrac{\overline{a} \vdash \sigma_{1i}' = \sigma_{2i}' \Rightarrow \psi_i \quad i = 1..n}{\overline{a} \vdash T\ \overline{\sigma}_1'^{1..n} \leq T\ \overline{\sigma}_2'^{1..n} \Rightarrow (\sqcup_{1..n}\psi_i)}\ \text{S-CON}$$

Figure 5: Pre-matching and Pre-substitution

For example, S-ANY shows that pre-subsumption can always return the empty substitution. However, if the program type checks, there will always be some "best" substitution that contains the most information, and that is what our algorithm uses.

Even if the program does not type check, pre-subsumption still returns a substitution, but in this case, there is no "best" substitution. For example, both judgements

$$a \vdash (a, a) \leq (\sigma_{id}, \texttt{Int}) \Rightarrow [a \mapsto \sigma_{id}]$$

$$a \vdash (a, a) \leq (\sigma_{id}, \texttt{Int}) \Rightarrow [a \mapsto \texttt{Int}]$$

are derivable. This behavior is not problematic—even though pre-subsumption does not detect the discrepancy between the types, the final subsumption check ($\vdash \psi_u(\sigma) \leq \rho'$) will fail. Specifying the system in this manner is slightly simpler than eagerly detecting type errors during pre-substitution. Furthermore, this decision simplifies the implementation of our algorithm: pre-subsumption and pre-matching can be easily implemented as pure functions that do not interfere with unification or box filling.

Concerning expressiveness, the rules SA-ARG, SA-RES and SA-UNB all are strictly better than the original smart-application rule, SMART-APP. These rules provide more checked information (through unboxing) when examining the arguments u_i or the result type ρ'. However, although SA-UNB appears more powerful SA-ARG and SA-RES, it is not a complete win—there are obscure terms that are typeable with SA-ARG or SA-RES that are not typeable with SA-UNB. But because we believe that SA-UNB behaves better in the common cases, and we find its lack of bias attractive, we have included this rule in our implementation.

6.3 Recovering completeness

Somewhat surprisingly, our algorithm extended with SA-UNB is not complete. The problem arises because, at a let-binding, the specification allows the let-bound identifier to be given its most general type, *but it also allows it to be given a less general type*. Unfortunately, with SA-UNB, there exist programs that will type check with the less general type (according to the subsumption relation), but not with the more general one! For example:

$$\begin{array}{ll} \vdash & \texttt{let}\, f = \backslash x\,.\,[\,] \\ & \texttt{in}\, f\ (\backslash g\,.\,(g\ 3, g\ \texttt{True})) : [(\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})] \end{array}$$

If f is assigned its most general type, $\forall b_1 b_2\,.\,b_1 \to [b_2]$, the body of the let-binding is untypeable. On the other hand the specification can assign the type $\forall a_1\,.\,a_1 \to [a_1]$ to the function, match a_1 with the polymorphic context type (via ψ_0 in SA-UNB), and succeed in checking the argument against the known type $(\forall a\,.\,a \to \texttt{Int}) \to (\texttt{Int}, \texttt{Int})$.

The cause of the completeness problem is that a less-general type may induce a different "sharing" of variables between argument and result types, leading to more possibilities for unboxing polymorphic information. Since the distinction between variables appearing in the argument and the result types appears in SA-UNB, SA-RES and SA-ARG but not in SMART-APP, the first three rules have this problem but not the last. One could, in principle, imagine a more elaborate algorithm to fix this problem, but such an algorithm would certainly be more involved than ours.

Now that we understand the problem, there is an obvious solution: modify the typing rules to specify that `let` generalisation must infer the most general type (in the Hindley-Milner sense). Such a solution is not new in type inference systems: similar ideas were used by Leroy and Mauny for the typing of dynamics in ML [12], and by Garrigue and Rémy in their extension of ML with semi-explicit first-class polymorphism [3]. In particular, one would replace LET-I in Figure 2 with the following rule:

$$\frac{\Gamma \vdash u : \boxed{\rho} \qquad \Gamma, x{:}\overline{\Gamma}(\rho) \vdash t : \rho' \qquad \forall \rho_0\ .\ \text{ If } \Gamma \vdash u : \boxed{\rho_0} \text{ then } \vdash \overline{\Gamma}(\rho) \leq_{HM} \overline{\Gamma}(\rho_0)}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \rho'}\ \text{LET-IP}$$

where $\overline{\Gamma}(\rho)$ is just a shorthand for the generalisation of ρ over its free type variables that do not occur in Γ. Rule LET-IP has the effect that all bindings in the environment have unique types for any possible typing derivation. However, we have not yet carried out proofs for this solution but we believe that our algorithm (which remains exactly the same) is complete for this specification.

7. Discussion

Contravariance and subsumption Why did we choose invariance when performing subsumption on function types (rule F1)? Suppose we had used contravariance. Now recall that Principle 2 requires that $\vdash \forall a\,.\,a \to \texttt{Int} \leq \sigma \to \texttt{Int}$. To form this judgement, SPEC instantiates the left-hand type with $\boxed{\sigma}$, and considers $\vdash \boxed{\sigma} \to \texttt{Int} \leq \sigma \to \texttt{Int}$. If F1 required contravariance for the function argument, we would need $\vdash \sigma \leq \boxed{\sigma}$, which would in turn require the following rule:

$$\frac{\vdash \sigma' \sim \boxed{\sigma}}{\vdash \sigma' \leq \boxed{\sigma}}\ \text{SBOXY-WRONG}$$

Alas, SBOXY-WRONG is incompatible with the simple inference algorithm we have in mind, because it overlaps with rule SPEC. In some situations the "right" thing is to apply SBOXY-WRONG, while in others one must apply SPEC. To see an example of the latter, consider that it must definitely be the case that $\vdash \forall a\ .\ a \leq \boxed{\texttt{Int}}$!

In short, to obtain complete type inference with a syntax-directed, search-free algorithm, we cannot use SBOXY-WRONG, and that in turn means that we cannot have both Principle 2 and contravariance on function arguments. In practice, argument contravariance does not seem very important, whereas Principle 2 seems vital; hence our choice.

Abstraction In our system, the environment Γ contains only *vanilla* types, not boxy types. This choice means that η-expansion may render a typeable program untypeable. For example:

$$f : (\texttt{Int} \to \sigma_{id}) \to \texttt{Int} \not\vdash \backslash x\,.\,f\ x : \boxed{(\texttt{Int} \to \sigma_{id}) \to \texttt{Int}}$$

because it would require x to enter the environment with a boxy type. In general, boxes in the environment would allow information from the occurrences of a variable to propagate to the type of the abstraction that binds that variable.

There is no fundamental obstacle to allowing Γ to contain boxy types, provided that we ensure that the boxes are filled in. For example, the ABS1 rule should be replaced by two rules:

$$\frac{x \in \mathit{fv}(t) \qquad \Gamma, x : \sigma_1' \vdash^{poly} t : \sigma_2'}{\Gamma \vdash (\backslash x\,.\,t) : \sigma_1' \to \sigma_2'}\ \text{ABS1A} \qquad \frac{x \notin \mathit{fv}(t) \quad \vdash \sigma_1' \sim \boxed{\sigma_1} \qquad \Gamma \vdash^{poly} t : \sigma_2'}{\Gamma \vdash (\backslash x\,.\,t) : \sigma_1' \to \sigma_2'}\ \text{ABS1B}$$

Our η-expanded example is typeable with ABS1A. But, note the condition $x \in \mathit{fv}(t)$. If x occurs in t (ABS1A), then we can figure out x's type from its occurrences in t. However, if x is not mentioned in t (in rule ABS1B), then any boxes in x's type, which represent its inferred parts, should be filled in with monotypes, as in the normal ABS1 rule.

8. Related work

Extensions to Hindley-Milner type inference This paper follows a series of papers that augment the HM type system with higher-rank and impredicative polymorphism, while remaining based on first-order unification. Many systems [17, 15, 6] retain the stratification between monotypes τ and polytypes σ. They support first-class polymorphism by embedding polymorphic types inside type constructors. Constructor introductions and eliminations mark the locations where type abstraction and application is necessary. In contrast, our vanilla types are the full types of System F.

Garrigue and Rémy's extension of ML with higher-rank polymorphism [3] embeds polytypes inside monotypes, eliminating the need to predeclare type schemes. Their types mark whether polytypes are annotated or inferred. Other than type signatures, they do not use any contextual information. Only annotated polytypes are allowed to be instantiated, at only marked locations.

The impressive ML^{F} language of Le Botlan and Rémy [9] supports impredicativity by extending polytypes with equality and generalization constraints. Their language of types is *richer* than System F or our language. Consequently, their language includes principal types for terms that have no principal type in System F. Furthermore, we are aware of some situations that require fewer annotations in ML^{F} than in our language. For example, ML^{F} has the property that if $t_1\ t_2$ type checks, then $apply\ t_1\, t_2$ type checks without annotation—but this property is not true here.

However, there are two reasons one might prefer our system to ML^{F}. We believe that our system is easier to add to existing compilers, in particular, those that use typed intermediate languages based on System F [24]. Although Leijen and Löh [11] have developed a translation from ML^{F} to System F, our translation (stripping boxes) is far simpler. Furthermore, because all of our types have a simple correspondence to System F types, we believe that our system is easier for current ML and Haskell programmers to understand.

Stratified type inference In parallel with our efforts, Rémy designed a two-phase approach to type inference for higher-rank and impredicative polymorphism called $\mathsf{F}^{?}_{\mathsf{ML}}$ [21]. The first phase infers the "shape" of the type of each variable in the program, where "shape" means the exact location of all quantifiers and the type variables they bind. Once shapes are known, only monotypes need be "guessed" by the second phase, which is done using ordinary unification. This division separates the mechanisms for propagating local type information (which must be done in a syntax-directed way) from the underlying first-order type inference. As a result, the two separate components of the type system may be thought about independently—but of course, both must be understood together to understand whether a program should type check.

Regarding expressivity, our system more aggressively propagates known polytype information than $\mathsf{F}^{?}_{\mathsf{ML}}$, for two reasons. First, boxy types and the unbiased smart application rule can more precisely express the local flow of polytype information than the shape inference algorithm of $\mathsf{F}^{?}_{\mathsf{ML}}$, although it is possible that a more sophisticated shape inference algorithm could capture this behavior. Second, the separation between shape propagation and type inference in $\mathsf{F}^{?}_{\mathsf{ML}}$ means that shape propagation can not take advantage of polymorphic types inferred by type inference (through generalization). In contrast, our system adds the inferred polymorphic type of the right-hand side of the let into the context as known type information before checking the body of the let (see rule LET-I). Therefore, our system includes a significant source of known polytype information that $\mathsf{F}^{?}_{\mathsf{ML}}$ does not. As a result, Rémy proposes incremental elaboration: the type of each top level definition is completely determined before continuing to the next one. However, this strategy treats top-level definitions differently than internal binding.

Another, more subtle difference between the systems is that $\mathsf{F}^{?}_{\mathsf{ML}}$ better supports a relation between polytypes called type containment [14]. In $\mathsf{F}^{?}_{\mathsf{ML}}$, shape propagation infers impredicative instantiations, while the subsumption relation used during type inference is a predicative version of type containment. (Full type containment is known to be undecidable [25].) In contrast, our system must do both simultaneously, so it infers impredicative instantiation at the expense of type containment. (Recall that we use invariance instead of contravariance for the argument component of function types.) However, it is not clear how much of an advantage this is to $\mathsf{F}^{?}_{\mathsf{ML}}$. Rémy remarks that even though ML^{F} does not support type containment, it has not been a problem in practice [21].

Because of these differences, there are many programs that type check in one system but not the other. We regard stratified type inference and boxy types as two alternative approaches to the question of how to exploit programmer-supplied type annotations. The two approaches feel different, but their expressive power is similar; it is too early to say which is superior, if indeed either is.

Local type inference Pierce and Turner [19] coined the term "Local Type Inference" to refer to a partial inference technique for a language with bounded, impredicative quantification and higher-rank types. (They attribute the original idea to John Reynolds.) They rejected unification entirely and based their type system on two ideas: local type argument synthesis and bidirectional propagation. Similar to our SMART-APP, local type argument synthesis infers the type argument to a polymorphic function by examining the types of its arguments. Bidirectional type checking operates in one of two modes: inference and checking. A subsequent development, Colored Local Type Inference (CLTI), by Odersky and Zenger [16], reformulated bidirectional checking for $F_{\leq}$ so that the *type* and not the *judgment form* describes the direction in which type information flows. Their colours are an inspiration for our "boxy" types, although our system has many major differences from theirs.

Most importantly, the distinction between "synthesized" and "inherited" type information—the colours in CLTI—is different from the distinction made between "inferred" and "checked" type information by our boxes. In CLTI, the colours trace the flow of information, either from the leaves to the root of the derivation or vice versa. So variables always have "synthesized" types, whereas in our system, they have "checked" types. Another difference is that the colours in CLTI types may nest, but our boxes do not.

Although type argument synthesis and bidirectional propagation provide an impressive amount of type inference, the resulting language is difficult for ML and Haskell programmers to use, because the lack of unification means that many programs require type annotations. Hoysoya and Pierce [4] note a few such situations. Although there is folklore about combining bidirectional propagation with HM inference, there is relatively little published work that describes such systems [1, 10].

9. Conclusions and further work

We have presented the first type system for impredicative polymorphism that is a conservative extension of the standard Hindley-Milner system, and can be implemented using a modest extension of classical unification-based type inference. Against these advantages, there are two obvious criticisms one could make. First, the system is somewhat complex. Second, although the type system is not an algorithm, it is carefully designed with an algorithm in mind: boxy types have no *logical* role, and instead serve to constrain the typeable programs to ones that are also inferable.

Although the system is guided by algorithmic intuitions, it is much simpler than the algorithm itself. Furthermore, although the system *can* be implemented using the algorithm we give, it could perhaps also be implemented in other ways, such as constraint generation. There is a real gain from separating specification (even an algorithmically-guided one) from implementation.

The type system is arguably too complicated for Joe Programmer to understand, but that is true of many type systems, and perhaps it does not matter too much: in practice, Joe Programmer usually works by running the compiler repeatedly, treating the compiler as the specification of the type system. Indeed, a good deal of the complexity of the type system (especially Section 6) is there to accommodate programs that "ought" to work, according to our understanding of Joe's intuitions. Nevertheless, a precise specification, such as the one we give, is very valuable because it tells compiler writers what to do. Even if Joe does not fully understand the type system, it is reasonable to expect compiler writers to do so.

We have a complete, downloadable implementation of the system described in this paper, including SA-UNB described in Section 6.2, embodied in the Glasgow Haskell Compiler. We have had no reports of unexpected behaviour, which suggests that the enhancements do not trip up programmers who do not employ them. We do not have sufficient experience to report one way or the other on the claims about programmer intuitions. However, we hope to better gauge the trade offs between the burden of user annotations and user predictability. Some users may prefer to write more annotations in exchange for a simpler specification of where they are necessary. To that end, we also hope to determine more properties of programs that do and do not need annotation.

In particular, we plan to explore how local transformations affect the typability of terms. For example, we have already discussed how η-expansion can have beneficial and detrimental effects. However, even if a local transformation causes a term to fail to type check, typability may always be recovered through annotation. The System F embedding in Section 4 provides a simple specification of annotations that are guaranteed to be sufficient. We also intend to explore variations of this type system, some of which we mentioned in Sections 6 and 7. Other variations follow from the nontrivial interaction between our system and ML-style references.

More generally, we believe that type systems will increasingly embody a blend of type inference and programmer-supplied type annotations: higher-rank types and impredicativity are examples of this trend, and there are plenty of others, such as polymorphic recursion, GADTs [18] or subtyping. Giving a precise, predictable and implementable specification of these blended type systems is a new challenge. Boxy types are a powerful tool in this respect and one that we hope to use again.

Acknowledgments We would like to thank Didier Rémy, Arthur Charguéraud, Jeffrey Vaughan, Steve Zdancewic, and the ICFP referees, for their helpful comments. We also thank Andres Löh, who helped to refine a precursor to the system we describe here. This work was partially supported by National Science Foundation grant CCF-0347289.

References

[1] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, pages 379–405, 1991.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York, 1982. ACM Press.

[3] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Journal of Information and Computation*, 155:134–169, 1999.

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

[5] *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia, September 2005. ACM.

[6] Mark P. Jones. First-class polymorphism with type inference. In *POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 483–496, Paris, France, 1997.

[7] AJ Kfoury and JB Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda calculus. In *ACM Symposium on Lisp and Functional Programming*, pages 196–207. ACM, Orlando, Florida, June 1994.

[8] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, January 2003. ACM Press.

[9] D Le Botlan and D Rémy. MLF: raising ML to the power of System F. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 27–38, Uppsala, Sweden, September 2003. ACM.

[10] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[11] Daan Leijen and Andres Lh. Qualified types for MLF. In ICFP05 [5], pages 144–155.

[12] X Leroy and M Mauny. Dynamics in ML. In RJM Hughes, editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, Boston, 1991. Springer Verlag.

[13] R Milner. A theory of type polymorphism in programming. *JCSS*, 13(3), December 1978.

[14] JC Mitchell. Coercion and type inference. In *ACM POPL*, pages 175–185. January 1984.

[15] M Odersky and K Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 54–67. ACM, St Petersburg Beach, Florida, January 1996.

[16] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, London, January 2001. ACM.

[17] N Perry. *The implementation of practical functional programming languages*. Ph.D. thesis, Imperial College, London, 1991.

[18] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, Portland, Oregon, 2006. ACM Press.

[19] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego, January 1998. ACM.

[20] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[21] Didier Rémy. Simple, partial type inference for System F, based on type containment. In ICFP05 [5], pages 130–143.

[22] Ken Shan. Sexy types in action. *SIGPLAN Notices*, 39(5):15–22, May 2004.

[23] Mark Shields and Simon Peyton Jones. Lexically scoped type variables. Microsoft Research, 2002.

[24] D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, and P Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 181–192. ACM, Philadelphia, May 1996.

[25] J Tiuryn and P Urzyczyn. The subtyping problem for second order types is undecidable. In *Proc IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 74–85, 1996.

[26] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity, Technical Appendix. Technical Report MS-CIS-05-23, University of Pennsylvania, April 2006.