FPH: First-class Polymorphism for Haskell

Declarative, Constraint-free Type Inference for Impredicative Polymorphism

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

Simon Peyton Jones
Microsoft Research
simonpj@microsoft.com

Abstract

Languages supporting polymorphism typically have ad-hoc restrictions on where polymorphic types may occur. Supporting "first-class" polymorphism, by lifting those restrictions, is obviously desirable, but it is hard to achieve this without sacrificing type inference. We present a new type system for higher-rank and impredicative polymorphism that improves on earlier proposals: it is an extension of Damas-Milner; it relies only on System F types; it has a simple, declarative specification; it is robust to program transformations; and it enjoys a complete and decidable type inference algorithm.

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 this program fragment[1]:

```
($)   :: forall a b. (a->b) -> a -> b
runST :: forall r. (forall s. ST s r) -> r
foo   :: forall s. Int -> ST s Int

...(runST $ foo 4)...
```

Here `($)`, whose type is given, is the apply combinator, often used by Haskell programmers to avoid writing parentheses.[2] From a programmer's point of view there is nothing very complicated about this program, yet it goes well beyond the traditional Damas-Milner type system (Damas and Milner 1982), by using two distinct forms of first-class polymorphism:

- `runST` takes an argument of polymorphic type—`runST` has a *higher-rank* type.
- The quantified type variable `a` in the type of `($)` is instantiated to the polymorphic type $\forall s.\texttt{ST}\ s\ \texttt{Int}$. Allowing the instantiation of quantified type variables with polytypes is called *impredicative* polymorphism.

Our goal, which we share with other authors (Le Botlan and Rémy 2003; Leijen 2008a), is to make such programs "just work" by lifting the restrictions imposed by the Damas-Milner type system.

Although there are several competing designs with the same general goal, the design space is now becoming clear, so this paper is not simply "yet another impenetrable paper on impredicative polymorphism". We give a detailed comparison in Section 7, but meanwhile the distinctive feature of our system is this: rather than maximizing expressiveness or minimizing implementation complexity, we focus on programmer accessibility by *minimizing the complexity of the specification*. More specifically, we make the following contributions:

- We describe and formalize a new type system, FPH, based on System F, capable of expressing impredicative polymorphism (Section 3). We show that FPH can express all of System F (Section 3.4).
- FPH is unusually small and simple for its expressive power. It can be explained informally in a few paragraphs (Section 2), and in particular has the following delightfully simple rule for when a type annotation is required: *a type annotation may be required only for a* `let`*-binding or* λ*-abstraction that has a non-Damas-Milner type* (Section 2.2). For example, a nested function call, such as `(f (g x) (h (t y)))`, may involve lots of impredicative instantiation, but never requires a type annotation.
- We give a syntax-directed variant of the type system (Section 4), and prove it sound and complete with respect to the earlier declarative rules.
- We have a sound and complete inference algorithm for FPH, which we sketch in Section 5. Internally, this implementation uses type schemes with bounded quantification in the style of ML^F (Le Botlan and Rémy 2003), but this internal sophistication is never shown to the programmer; it is simply the mechanism used by the implementation to support the simple declarative specification.

Our system is fully compatible with the standard idea of propagating annotations via a so-called bidirectional type system. We discuss this and other design variants in Section 6. Finally, with the scaffolding now in place, Section 7 amplifies our opening remarks by showing in detail how the various current designs relate to each other.

Auxiliary material and proofs can be found in the first author's dissertation (Vytiniotis 2008).

[1] We use Haskell syntax, and will often prefix examples with type signatures for any functions used in the fragment.

[2] The example is equivalent to `(runST (foo 4))`.

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

2. Type inference for first-class polymorphism

To describe the main difficulty with first-class polymorphism, we first distinguish between Damas-Milner types (types permitting only top-level quantification) and *rich* types (types with $\forall$ quantifiers under type constructors). For example, `Int` $\rightarrow$ `Int` and $\forall a\,.\,a \rightarrow a$ are Damas-Milner types; but `Int` $\rightarrow [\forall a\,.\,a \rightarrow a]$ and $\forall a\,.\,(\forall b\,.\,b) \rightarrow [a]$ are rich types.

Both forms of first-class polymorphism (higher-rank and impredicative) result in a lack of principal types for expressions: a single expression may be typeable with two or more *incomparable* types, where neither is more general than the other. As a consequence, type inference cannot always choose a single type and use it throughout the scope of a `let`-bound definition.

1. With higher-rank polymorphism, functions that accept polymorphic arguments may be typed with two or more incomparable System F types. For example, consider the function `f` below:

   ```
   f get = (get 3, get True)
   ```

   It is clear that `get` must be assigned a polymorphic type in the environment, since we must be able to apply it to both `3` and `True`. But what is the exact type of `f`? For example, both $(\forall a\,.\,a \rightarrow a) \rightarrow (\texttt{Int}, \texttt{Bool})$, and $(\forall a\,.\,a \rightarrow \texttt{Int}) \rightarrow (\texttt{Int}, \texttt{Int})$ are valid types for `f`, but there exists no *principal* type for `f` such that all others follows from it by a sequence of instantiations and generalizations. Previous work has suggested that the programmer should be required to supply a *type annotation* for any function argument that must be polymorphic, so that the type of `f` is no longer ambiguous—the above code would fail to type check, but the annotation below would fix the problem:

   ```
   f (get :: forall a. a->a) = (get 3, get True)
   ```

2. The presence of impredicative instantiation of type variables leads to a second case of incomparable types. For example:

   ```
   choose :: forall a. a -> a -> a
   id     :: forall b. b -> b
   g = choose id
   ```

   In a traditional Damas-Milner type system, `g` would get the type $\forall b\,.\,(b \rightarrow b) \rightarrow (b \rightarrow b)$. However, if `choose` may be instantiated with a polymorphic type, `g` is also typeable with the incomparable type $(\forall b\,.\,b \rightarrow b) \rightarrow (\forall b\,.\,b \rightarrow b)$. This problem has been identified in the ML^F work and circumvented by *extending* the type language to include instantiation constraints. This extended type language can express a principal type for `g`, namely $\forall(a \geq \forall b\,.\,b \rightarrow b)\,.\,a \rightarrow a$. However, if one wants to remain within the type language of System F, the type system must specify which of these incomparable types is assigned to `g`. In FPH, `g` is typeable with its best Damas-Milner type $\forall b\,.\,(b \rightarrow b) \rightarrow (b \rightarrow b)$, but the type $(\forall b\,.\,b \rightarrow b) \rightarrow (\forall b\,.\,b \rightarrow b)$ is also available by using an explicit type signature, as follows:

   ```
   g = choose id :: (forall b.b->b) -> (forall b.b->b)
   ```

The focus of this paper is on impredicativity (item (2) above), since earlier work has essentially solved the question of higher-rank types (Peyton Jones et al. 2007). The core type system we present in Section 3 therefore does not support λ-abstractions with higher-rank types, focusing exclusively on impredicative instantiations. A practical system must accommodate higher-rank types as well, and we describe how previous work can be adapted to our setting in Sections 3.4 and 6.1.

2.1 Marking impredicative instantiation

We present a flavor of FPH in this section, and use several examples to motivate its design principles. Consider this program fragment:

```
str :: [Char]
ids :: [forall a. a->a]
length :: forall b. [b] -> Int

l1 = length str
l2 = length ids
```

First consider type inference for `l1`. The polymorphic `length` returns the length of its argument list, where the type `[b]` means "list of `b`". In the standard Damas-Milner type system, one instantiates the type of `length` with `Char`, so that the occurrence of `length` has type `[Char]` $\rightarrow$ `Int`, which marries up correctly with `length`'s argument, `str`. In Damas-Milner, a polymorphic function can only be instantiated with monotypes, where a monotype τ is a type containing no quantification (we use $[\cdot]$ for lists):

$$\tau ::= a \mid \tau_1 \rightarrow \tau_2 \mid [\tau]$$

This Damas-Milner restriction means that `l2` is untypeable, because here we must instantiate `length` with $\forall a.a \rightarrow a$. We cannot simply lift the Damas-Milner restriction, because that directly leads to the problem identified at the start of this section: different choices can lead to incomparable types. However, `l2` also shows that there are benign uses of impredicative instantiation. Although we need an impredicative instantiation to make `l2` type check, there is no danger here—the type of `l2` will always be `Int`. It is only when a `let`-binding can be assigned two or more incomparable types that we run into trouble.

Our idea is to mark impredicative instantiations so that we know when an expression may be typed with different incomparable types. Technically, this means that we instantiate polymorphic functions with a form of type τ' that is more expressive than a mere monotype, but less expressive than an arbitrary polymorphic type:

$$\begin{array}{lcl} \tau' & ::= & a \mid \tau'_1 \rightarrow \tau'_2 \mid [\tau'] \mid \boxed{\sigma} \\ \sigma & ::= & \forall a.\sigma \mid a \mid \sigma \rightarrow \sigma \mid [\sigma] \end{array}$$

Unlike a monotype τ, a *boxy monotype* τ' may contain quantification, but only inside a box, thus $\boxed{\sigma}$. **Idea 1** is this: a polymorphic function is instantiated with boxy monotypes. A boxy type marks the place in the type where "guessing" is required to fill in a type that makes the rest of the typing derivation go through.

Now, when typing `l2` we may instantiate `length` with $\boxed{\forall a.a \rightarrow a}$. Then the application `length ids` has a function expecting an argument of type $[\boxed{\forall a.a \rightarrow a}]$, applied to an argument of type $[\forall a.a \rightarrow a]$. Do these types marry up? Yes, they do, because of **Idea 2**: when comparing types, discard all boxes. The sole purpose of boxes is to mark polytypes that arise from impredicative instantiations. That completes the typing of `l2`.

Boxes are ignored when typing an application, but they play a critical role in `let` polymorphism. **Idea 3** is this: to make sure that there is no ambiguity about guessed polytypes, the type environment contains no boxes. Let us return to the example `g = choose id` given above. If we instantiated `choose` with the boxy monotype $\boxed{\forall a.a \rightarrow a}$, the application `(choose id)` would marry up fine, but its result type would be $\boxed{\forall a.a \rightarrow a} \rightarrow \boxed{\forall a.a \rightarrow a}$. However, **Idea 3** prevents that type from entering the environment as the type for `g`, so this instantiation for `choose` is rejected. If we instead instantiate `choose` with $c \rightarrow c$, the application again marries up (this time by instantiating the type of `id` with c), so the application has type $(c \rightarrow c) \rightarrow c \rightarrow c$, which can be generalized and then enter the environment as the type of `g`. This type is the principal Damas-Milner type of `g`—all Damas-Milner types for `g` are also available without annotation. What we have achieved effectively is that, instead of having two or more incomparable types for `g`, we have allowed only a subset of the possible System F typing derivations for `g` that does admit a principal type.

However, if the programmer actually wanted the other, rich, type for g, she can use a type annotation:

```
g = choose id :: (forall b.b->b) -> (forall b.b->b)
```

Such type annotations use **Idea 2**—when typing an annotated expression $e :: \sigma$, ignore boxes on e's type when comparing with σ (which is box-free, being a programmer annotation). Now we may instantiate `choose` with $\boxed{\forall a.a \to a}$, because the type annotation is compatible with the type of `(choose id)`, $\boxed{\forall a.a \to a} \to \boxed{\forall a.a \to a}$.

2.2 Expressive power

As we have seen, a type annotation may be required on a `let`-bound expression, but annotations are never required on function applications, even when they are nested and higher order, or involve impredicativity. Here is the example from the Introduction, with some variants:

```
runST  :: forall a. (forall s. ST s a) -> a
app    :: forall a b. (a -> b) -> a -> b
revapp :: forall a b. a -> (a -> b) -> b
arg    :: forall s. ST s Int

h0 = runST arg
h1 = app runST arg
h2 = revapp arg runST
```

All definitions `h0`, `h1`, `h2` are typeable without annotation because, in each case, the return type is a (non-boxy) monotype `Int`.

Actually, we have a much more powerful guideline for programmers, which does not even require them to think about boxes:

> **Annotation Guideline.** Write your programs as you like, without type annotations at all. Then you are required to annotate only those `let`-bindings and λ-abstractions that you want to be typed with rich types.

For instance, for a term consisting of applications and variables to be `let`-bound (without any type annotations), it *does not matter* what impredicative instantiations may happen to type it, provided that the result type is an ordinary Damas-Milner type! For example, the argument `choose id` to the function `f` below involves an impredicative instantiation (in fact for both `f` *and* `choose`), but no annotation is required whatsoever:

```
f :: forall a. (a -> a) -> [a] -> a
g = f (choose id) ids
```

In particular `choose id` gets type $\boxed{\forall a.a \to a} \to \boxed{\forall a.a \to a}$. However, `f`'s arguments types can be married up using **Idea 2**, and its result type (ignoring boxes) is a Damas-Milner type ($\boxed{\forall a.a \to a}$), and hence no annotation is required for `g`.

Since the Annotation Guideline does not require the programmer to think about boxes at all, why does our specification use boxes? Because the Annotation Guideline is conservative: it guarantees to make the program typeable, but it adds more annotations than are necessary. For example:

```
f' :: forall a. [a] -> [forall b. b -> b]
g' = f' ids
```

Notice that the rich result type `[forall b. b -> b]` is non-boxy, and hence no annotation is required for `g'`. In general, even if the type of a `let`-bound expression is rich, if that type does not result from impredicative instantiation (which is the common case), then no annotations are required. Boxes precisely specify what "that type does not result from impredicative instantiation" means. Nevertheless, a box-free specification is an attractive alternative design, as we discuss in Section 6.3.

Types	σ	::=	$\forall \overline{a}.\rho$
	ρ	::=	$\tau \mid \sigma \to \sigma$
	τ	::=	$a \mid \tau \to \tau$
Boxy Types	σ'	::=	$\forall \overline{a}.\rho'$
	ρ'	::=	$\tau' \mid \sigma' \to \sigma'$
	τ'	::=	$a \mid \boxed{\sigma} \mid \tau' \to \tau'$
Environments	Γ	::=	$\Gamma, (x{:}\sigma) \mid \cdot$

Figure 1: Syntax

2.3 Limitations of FPH

Although the FPH system, as we have described it so far, is expressive, it is also somewhat conservative. It requires annotations in a few instances, even when there is only one type that can be assigned to a `let`-binding, as the following example demonstrates.

```
f :: forall a. a -> [a] -> [a]
ids :: [forall a. a -> a]

h1 = f (\x -> x) ids            -- Not typeable
h2 = f (\x -> x) ids :: [forall a. a->a]  -- OK
```

Here `f` is a function that accepts an element and a list and returns a list (for example, `f` could be `cons`). Definition `h1` is not typeable in FPH. We can attempt to instantiate `f` with $\boxed{\forall a.a \to a}$, but then the right hand side of `h1` has type $[\boxed{\forall a.a \to a}]$, and that type cannot enter the environment. The problem can of course be fixed by adding a type annotation, as `h2` shows.

You may think that it is silly to require a type annotation in `h2`; after all, `h1` manifestly has only one possible type! But suppose that `f` had type $\forall ab.a \to b \to [a]$, which is a more general Damas-Milner type than the type above. With this type for `f`, our example `h1` now has *two incomparable types*, namely $[\forall a.a \to a]$ as before, and $\forall a.[a \to a]$. Without any annotations we presumably have to choose the same type as the Damas-Milner type system would; and that might make occurrences of `h1` ill typed. In short, making the type of `f` more general (in the Damas-Milner sense) has caused definitions in the scope of `h1` to become ill-typed! This is bad; and that is the reason that we reject `h1`, requiring an annotation as in `h2`.

Requiring an annotation on `h2` may seem an annoyance to programmers, but it is this conservativity of FPH that results in a simple and declarative high-level specification. FPH allows `let`-bound definitions to enter environments with many different types, as is the case in the Damas-Milner type system.

3. Declarative specification of the type system

We now turn our attention to a systematic treatment of FPH, beginning with the basic syntax of types and environments in Figure 1. Types are divided into box-free types σ-, ρ-, and τ-types, and boxy types σ', ρ', and τ' types. Polymorphic types, σ and σ', may contain quantifiers at top-level, whereas ρ and ρ' types contain only nested quantifiers. The important difference between box-free and boxy types occurs at the monotype level. Following previous work by Rémy *et al.* (Garrigue and Rémy 1999; Le Botlan and Rémy 2003), τ' may include boxes containing (box-free) polytypes. As we discussed in Section 2.1, these boxes represent the places where "guessed instantiations" take place. Note that we do not include syntax for type constructors other than $\to$, as their treatment is very similar to the treatment of $\to$. The syntax of type environments, Γ,

directly expresses **Idea 3** in Section 2.1 by allowing only box-free types σ.

3.1 Typing rules

The declarative (i.e. not syntax-directed) specification of FPH is given in Figure 2. As usual, the judgement form $\Gamma \vdash e : \sigma'$ assigns the type σ' to the expression e in typing environment Γ. A non-syntactic invariant of the typing relation is that, in the judgement $\Gamma \vdash e : \forall \overline{a}.\rho'$, no box may intervene between a variable quantified inside ρ' and the occurrences of that variable. Thus, for example, ρ' cannot be of form $(\forall b.\boxed{b}) \to \texttt{Int}$, because the quantified variable b appears inside a box. The top-level quantified variables may, however, appear inside boxes.

The rules in Figure 2 are modest (albeit carefully-chosen) variants of the conventional Damas-Milner rules. Indeed rule VAR is precisely as usual, simply returning the type of a variable from the environment.

Rule APP infers a function type $\sigma'_1 \to \sigma'_2$ for e_1, infers a type σ'_3 for the argument e_2, and checks that the argument type matches the domain of the function type *modulo boxy structure*, implementing **Idea 2** of Section 2.1. This compatibility check is performed by *stripping* the boxes from σ'_1 and σ'_3, then comparing for equality. The notation $\lfloor \sigma' \rfloor$ denotes the non-boxy type obtained by discarding the boxes in σ':

Definition 3.1 (Stripping) *We define the strip function $\lfloor \cdot \rfloor$ on boxy types as follows:*

$$\begin{array}{lcll} \lfloor a \rfloor & = & a & \\ \lfloor \boxed{\sigma} \rfloor & = & \sigma & \\ \lfloor \sigma'_1 \to \sigma'_2 \rfloor & = & \lfloor \sigma'_1 \rfloor \to \lfloor \sigma'_2 \rfloor & \\ \lfloor \forall \overline{a}.\rho' \rfloor & = & \forall \overline{a}\overline{b}.\rho & \text{where } \lfloor \rho' \rfloor = \forall \overline{b}.\rho \end{array}$$

Stripping is also used in rule ANN, which handles expressions with explicit programmer-supplied type annotations. It infers a boxy type for the expression and checks that, modulo its boxy structure, it is equal to the type required by the annotation σ. In effect, this rule converts the boxy type σ'_1 that was inferred for the expression to a box-free type σ. If the annotated term is the right-hand side of a `let` binding `x = e::`σ, this box-free type σ can now enter the environment as the type of `x` (whereas σ' could not, by **Idea 3**).

Rule ABS infers types for λ-abstractions. It first extends the environment with a *monomorphic, box-free* typing $x : \tau$, and infers a ρ-type for the body of the function. Notice that we insist (syntactically) that the result type ρ both (a) has no top-level quantifiers, and (b) is box-free. We exclude top-level quantifiers (a) because we wish to attribute the same types as Damas-Milner for programs that are typeable by Damas-Milner, that is, we avoid "eager generalization" (Peyton Jones et al. 2007). Choice (b), that a λ-abstraction must return a box-free type, may require more programmer annotations, but turns out to permit a much simpler type inference algorithm. We return to this issue in Section 6.3.

Rule ABS is the main reason that the type system of Figure 2 cannot type all of System F, even with the addition of type annotations: ABS allows only abstractions of type $\tau \to \rho$, whereas System F has λ-abstractions of type $\sigma_1 \to \sigma_2$. Rule ABS is however just enough to demonstrate our approach to impredicative instantiation (the contribution of this paper), while previous work (Peyton Jones et al. 2007) has shown how to address this limitation. It is easy to combine the two, as we show in Section 3.4.

Following **Idea 3** of Section 2.1, rule LET first infers a *box-free* type σ for the right-hand side expression u, and then checks the body pushing the binder x with type σ in the environment.

Generalization (GEN) takes the conventional form, where $\overline{a}\#\Gamma$ means that $\overline{a}$ is disjoint from the free type variables of Γ. In this rule, note that the generalized variables $\overline{a}$ may appear inside boxes in ρ', so that we might, for example, infer $\Gamma \vdash e : \forall a.\boxed{a} \to a$.

Instantiation (INST) is conventional, but it follows **Idea 1** by allowing us to instantiate with a *boxy* monotype τ'. However, we need to be a little careful with substitution in INST: since ρ' may contain $\overline{a}$ inside boxes, a naive substitution might leave us with nested boxes, which are syntactically ill-formed. Hence, we define a form of substitution that preserves the boxy structure of its argument.

Definition 3.2 (Monomorphic substitutions) *We use letter φ for* monomorphic substitutions*, that is, φ denotes finite maps of the form $[\overline{a \mapsto \tau'}]$. We let $ftv(\varphi)$ be the set of the free variables in the range and domain of φ. We define the operation of applying φ to a type σ' as follows:*

$$\begin{array}{lcll} \varphi(a) & = & \tau' & \text{where } [a \mapsto \tau'] \in \varphi \\ \varphi(\boxed{\sigma}) & = & \boxed{\lfloor \varphi(\sigma) \rfloor} & \\ \varphi(\sigma'_1 \to \sigma'_2) & = & \varphi(\sigma'_1) \to \varphi(\sigma'_2) & \\ \varphi(\forall \overline{a}.\rho') & = & \forall \overline{a}.\varphi(\rho') & \text{where } \overline{a}\#ftv(\varphi) \end{array}$$

We write $[\overline{a \mapsto \tau'}]\sigma'$ for the application of the $[\overline{a \mapsto \tau'}]$ to σ'.

3.2 The subsumption rule

The final rule, SUBS, is tricky but important. Consider below:

Example 3.1 (Boxy instantiation)

```
head :: forall a. [a] -> a
h = head ids 3
```

Temporarily ignoring rule SUBS in Figure 2, `head ids` can get type $\boxed{\forall a.a \to a}$, and only that type. Hence, the application `(head ids) 3` cannot be typed. This situation would be rather unfortunate as one would, in general, have to use type annotations to extract polymorphic expressions out of polymorphic data structures. For example, programmers would have to write:

```
h = (head ids :: forall b. b -> b) 3
```

This situation would also imply that some expressions which consist only of applications of closed terms, and are typeable in System F, could not be typed in FPH.

Rule SUBS addresses these limitations. Rule SUBS modifies the types of expression in two ways with the relation $\preceq\sqsubseteq$, which is the composition of two relations, $\preceq$, and $\sqsubseteq$. The relation $\preceq$, called *boxy instantiation*, simply instantiates a polymorphic type within a box. The relation $\sqsubseteq$, called *protected unboxing*, removes boxes around monomorphic types and pushes boxes congruently down the structure of types. The most important rules of this relation are TBOX and REFL. The first simply removes a box around a monomorphic type, while the second ensures reflexivity. If a ρ' type contains only boxes with monomorphic information, then these boxes can be completely dropped along the $\sqsubseteq$ relation to yield a box-free type. Finally, nothece that the addition of arbitrary constructors is a straightforward adaptation of the rules for function types.

Because SUBS uses $\preceq\sqsubseteq$ instead of merely $\sqsubseteq$, `h` in Example 3.1 is typeable. When we infer a type for `head ids`, we may have the following derivation:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{head ids} : \boxed{\forall a.a \to a} \quad \boxed{\forall a.a \to a} \preceq \boxed{a \to a} \sqsubseteq a \to a}{\Gamma \vdash \texttt{head ids} : a \to a}\ \text{SUBS}}{\Gamma \vdash \texttt{head ids} : \forall a.a \to a}\ \text{GEN}$$

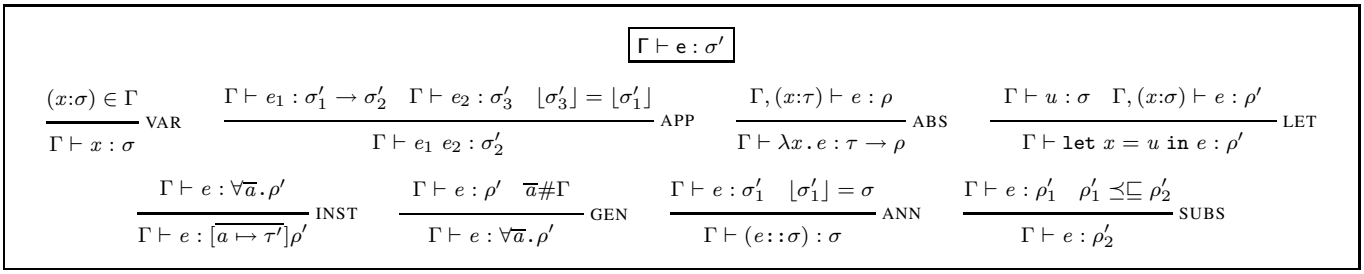

Figure 2: The FPH system

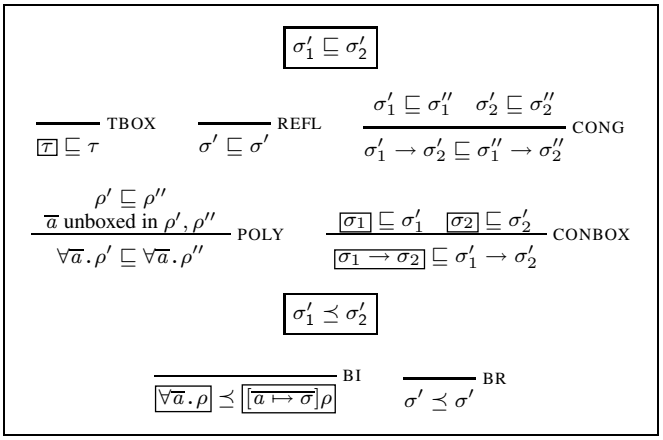

Figure 3: Protected unboxing and boxy instantiation relation

Therefore, no annotation is required on `h`. Incidentally, because the $\sqsubseteq$ relation can remove boxes around monomorphic types, it also follows that the definition

```
f = head ids
```

is typeable. More generally, we have the following lemma.

Lemma 3.2 *If* $\Gamma \vdash e : \boxed{\forall \overline{a}\,.\,\tau}$ *then* $\Gamma \vdash e : \forall \overline{a}\,.\,\tau$.

Proof: by rule BI we can instantiate $\forall \overline{a}\,.\,\tau$ with (monomorphic) fresh $\overline{a}$, use rule TBOX to strip boxes, and finally use rule GEN.

3.3 Properties

The FPH system is type safe with respect to the semantics of System F. The following lemma is an easy induction after observing that whenever $\sigma_1' \preceq \sqsubseteq \sigma_2'$, it is the case that $\vdash^{\mathsf{F}} \lfloor \sigma_1' \rfloor \leq \lfloor \sigma_2' \rfloor$, where $\vdash^{\mathsf{F}}$ is the System F *type instance relation*. The relation $\vdash^{\mathsf{F}}$ specifies typeability of an expression of one type with another type through a series of instantiations and generalizations, and is given by the rule below:

$$\frac{\overline{b} \# ftv(\forall \overline{a}\,.\,\rho)}{\vdash^{\mathsf{F}} \forall \overline{a}\,.\,\rho \leq \forall \overline{b}\,.\,[\overline{a \mapsto \sigma}]\rho}\ \text{FSUBS}$$

Lemma 3.3 *If* $\Gamma \vdash e : \sigma'$ *then* $\Gamma \vdash^{\mathsf{F}} e^{\flat} : \lfloor \sigma' \rfloor$, *where* $e^{\flat}$ *simply removes the type annotations from* e, *and* $\vdash^{\mathsf{F}}$ *is the typing relation of implicitly typed System F.*

Moreover, FPH is an extension of the Damas-Milner type system. The idea of the following lemma is that instantiation to τ' types always subsumes instantiation to τ types.

Lemma 3.4 (Extension of Damas-Milner) *Assume that* Γ *only contains Damas-Milner types and* e *is annotation-free. Then* $\Gamma \vdash^{\mathsf{DM}} e : \sigma$ *implies that* $\Gamma \vdash e : \sigma$.

We conjecture that the converse direction is also true, that is, unannotated programs in contexts that use only Damas-Milner types are typeable in Damas-Milner if they are typeable in FPH, but we leave this result as future work.

3.4 Higher rank types and System F

As we remarked in the discussion of rule ABS in Section 3.1, the system described so far deliberately does not support λ-abstractions with higher-rank types, and hence cannot yet express all of System F. For example:

Example 3.5

```
f   :: forall a. a -> [a] -> Int
foo :: [Int -> forall b.b->b]

bog = f (\x y ->y) foo
```

Here, `foo` requires the λ-abstraction `\x y -> y` to be typed with type $\texttt{Int} \to \forall b\,.\,b \to b$, but no such type can be inferred for the λ-abstraction, as it is not of the form $\tau \to \rho$. We may resolve this issue by adding a new syntactic form, the annotated λ-abstraction, thus $(\lambda x\,.\,e :: \sigma_1 \to \sigma_2)$. This construct provides an annotation for both argument (σ_1, instead of a monotype τ) and result (σ_2 instead of ρ). Its typing rule is simple:

$$\frac{\Gamma, (x{:}\sigma_1) \vdash e : \sigma_2' \quad \lfloor \sigma_2' \rfloor = \sigma_2}{\Gamma \vdash (\lambda x\,.\,e :: \sigma_1 \to \sigma_2) : \sigma_1 \to \sigma_2}\ \text{ABS-ANN}$$

With this extra construct we can translate any implicitly-typed System F term into a well-typed term in FPH, using the translation of Figure 4. This type-directed translation of implicitly typed System F is specified as a judgement $\Gamma \vdash^{\mathsf{F}} e_{\mathsf{F}} : \sigma \rightsquigarrow e$ where e is a term that type checks in our language. Notice that the translation requires annotations *only* on λ-abstractions that involve rich types[3].

A subtle point is that the translation may generate *open* type annotations. For example, consider the implicitly typed System F term below:

$$\vdash \lambda x\,.\,e : \forall a\,.(\forall b\,.\,b \to a) \to a$$

Translating this term using Figure 4 gives

$$\vdash (\lambda x\,.\,e :: (\forall b\,.\,b \to a) \to a)$$

Note that the type annotation mentions a which is nowhere bound. Although we have not emphasized this point, FPH already accommodates such annotations.

[3] Of course, it would be fine to annotate *every* λ-abstraction, but the translation we give generates smaller terms.

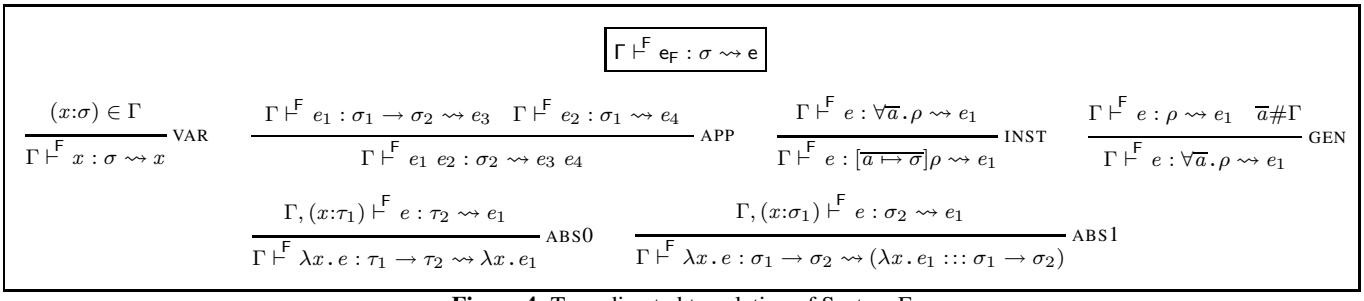

Figure 4: Type-directed translation of System F

The following theorem captures the essence of the type-directed translation.

Theorem 3.6 *If* $\Gamma \vdash^{\mathsf{F}} e : \sigma \rightsquigarrow e_1$ *then* $\Gamma \vdash e_1 : \sigma'$ *for some* σ' *such that* $\lfloor \sigma' \rfloor = \sigma$.

In practice, however, we do not recommend adding annotated λ-abstractions as a clunky new syntactic construct. Instead, with a bidirectional typing system we can get the same benefits (and more besides) from ordinary type annotations $e :: \sigma$, as we sketch in Section 6.1.

3.5 Predictability and robustness

A key feature of FPH is that it is simple for the programmer to figure out when a type annotation is required. We gave some intuitions in Section 2, but now we are in a position to give some specific results. The translation of System F to FPH of Section 3.4 shows that one needs only annotate `let`-bindings or λ-abstractions that must be typed with rich types. This is a result of combining Theorem 3.6 and Lemma 3.2.

For example, every applicative expression—one consisting only of variables, constants, and applications—that is typeable in System F is typeable in FPH without annotations. We began this paper with exactly such an example, involving `runST`, and it would work equally well if we had used reverse application instead of `$`.

Theorem 3.7 *If* e *is an applicative expression and* $\Gamma \vdash^{\mathsf{F}} e : \sigma$, *then* $\Gamma \vdash e : \sigma'$ *for some* σ' *with* $\lfloor \sigma' \rfloor = \sigma$.

It is easy to see this result by inspecting the rules of Figure 2. Functions may be instantiated with an arbitrary boxy type, but rule APP ignores the boxes.

Additionally, a `let`-binding can always be inlined at its occurrence sites. More precisely if $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ e : \sigma'$, then $\Gamma \vdash [x \mapsto u]e : \sigma'$. This follows from the following lemma:

Lemma 3.8 *If* $\Gamma \vdash u : \sigma$ *and* $\Gamma, (x{:}\sigma) \vdash e : \sigma'$ *then* $\Gamma \vdash [x \mapsto u]e : \sigma'$.

The converse direction cannot be true in general (although it is true for ML and ML^{F}) because of the limited expressive power of System F types, as we discussed briefly in Section 2. Let $\sigma_1 = (\forall b.b \to b) \to (\forall b.b \to b)$, $\sigma_2 = \forall b.(b \to b) \to b \to b$, $f_1 : \sigma_1 \to \texttt{Int}$, and $f_2 : \sigma_2 \to \texttt{Int}$. One can imagine a program of the form:

$$\ldots (f_1\ (\texttt{choose id})) \ldots (f_2\ (\texttt{choose id})) \ldots$$

which may be typeable, but it cannot be the case that: $\texttt{let}\ x = \texttt{choose id in} \ldots (f_1\ x) \ldots (f_2\ x) \ldots$ is typeable, as x can be bound with only one of the two incomparable types (in fact only with $\forall b.(b \to b) \to b \to b$).

However, notice that if an expression is typed with a box-free type at each of its occurrences in a context, it may be `let`-bound out of the context. For example, since λ-abstractions are typed with box-free types, if $\mathcal{C}[\lambda x.e]$ is typeable, where $\mathcal{C}$ is a multi-hole context, then it is always the case that $\texttt{let}\ f = (\lambda x.e)\ \texttt{in}\ \mathcal{C}[f]$ is typeable.

4. Syntax-directed specification

We now show how FPH may be implemented. The first step in establishing an algorithmic implementation is to specify a syntax-directed version of the type system, with the judgement $\Gamma \vdash^{\mathsf{sd}} e : \rho'$, where uses of the non-syntax-directed rules (SUBS, INST, and GEN) have been pushed to appropriate nodes inside the syntax-tree. Subsequently we may proceed with a low-level implementation of the syntax-directed system (Section 5). Our syntax-directed presentation appears in Figure 5.

Rule SDVAR instantiates the type of a variable bound in the environment, using the auxiliary judgement, $\vdash^{\mathsf{inst}} \sigma' \le \rho'$. The latter instantiates the top-level quantifiers of σ' to yield a ρ' type. However, we instantiate with boxes instead of τ' types, which is closer to the actual algorithm as boxes correspond to fresh "unification" variables.

Rule SDAPP deals with applications. It infers a type ρ' for the function, and uses $\preceq$ (Figure 3) and $\sqsubseteq^{\to}$ (a subset of $\sqsubseteq$) to expose an arrow constructor. The latter step is called *arrow unification*. Then SDAPP infers a ρ'_3 type for the argument of the application, generalizes over free variables that do not appear in the environment and checks that the result is more polymorphic (along the System F type instance) than the required type. Finally SDAPP instantiates the return type.

Rule SDABS uses a τ type for the argument of the λ-abstraction, and then forces the returned type ρ' for the body to be unboxed to a ρ-type using $\rho' \preceq \sqsubseteq \rho$. Finally, we consider all the free variables of the abstraction type that do not appear in the environment, and substitute them with arbitrary boxes. The returned type for the λ-abstraction is $\boxed{\overline{a \mapsto \boxed{\sigma}}}(\tau \to \rho)$.

This last step, of generalization and instantiation, is perhaps puzzling. After all rule ABS (in the declarative specification of Figure 2) seems to only force λ-abstractions to have box-free types. Here is an example to show why it is needed:

Example 4.1 (Impredicative instantiations in λ-abstractions) *The following derivation holds:* $\Gamma \vdash (\lambda x.x)\ \texttt{ids} : \boxed{[\forall a.a \to a]}$.

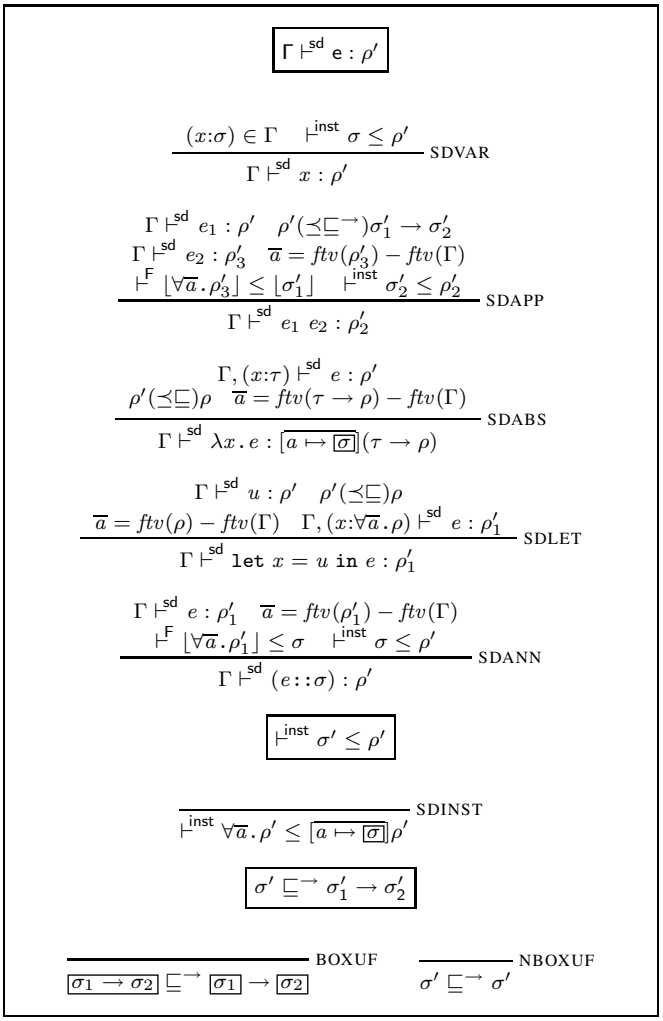

Figure 5: Syntax-directed Constrained Boxy Types system

To construct a derivation for Example 4.1 observe that we can instantiate $\lambda x.x$ with a polymorphic argument type, as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma, (x{:}a) \vdash x : a}{\Gamma \vdash \lambda x.x : a \to a}\ \text{ABS}}{\Gamma \vdash \lambda x.x : \forall a.\, a \to a}\ \text{GEN}}{\Gamma \vdash \lambda x.x : \boxed{[\forall a.\, a \to a]} \to \boxed{[\forall a.\, a \to a]}}\ \text{INST}$$

The use of GEN and INST are essential to make the term applicable to `ids` : $[\forall a.a \to a]$. The generalization and instantiation in SDABS ensure that GEN and INST are performed at each λ-abstraction, much as SDLET ensures that GEN is performed at each `let`-binding.

Rule SDLET is straightforward; after inferring a type for u which may contain boxes, we check that the boxes can be removed by $\preceq\sqsubseteq$ to get a ρ-type, which can subsequently be generalized and pushed in the environment.

Finally, rule SDANN infers a type ρ_1' for the expression e, generalizes over its free variables not in the environment, and checks that this type is more polymorphic than the annotations. As the final step, the annotation type is instantiated.

We can now establish the soundness of the syntax-directed system with respect to the declarative one.

Theorem 4.2 (Soundness of $\vdash^{\mathsf{sd}}$) *If* $\Gamma \vdash^{\mathsf{sd}} e : \rho'$ *then* $\Gamma \vdash e : \rho'$.

The proof is a straightforward induction over the derivation tree. The most interesting case is application which makes use of an auxiliary Lemma 4.3, given next, together with the fact that $\sqsubseteq^{\rightarrow}$ is a subset of $\sqsubseteq$.

Lemma 4.3 *If* $\Gamma \vdash e : \sigma_1'$ *and* $\vdash^{\mathsf{F}} \lfloor \sigma_1' \rfloor \le \lfloor \sigma_2' \rfloor$ *then* $\Gamma \vdash e : \sigma_3'$ *such that* $\lfloor \sigma_3' \rfloor = \lfloor \sigma_2' \rfloor$.

Conversely, the syntax-directed system is complete with respect to the declarative system, as the following theorem shows.

Theorem 4.4 (Completeness of $\vdash^{\mathsf{sd}}$) *If* $\Gamma \vdash e : \rho'$ *then* $\Gamma \vdash^{\mathsf{sd}} e : \rho_0'$ *such that* $\rho_0' \preceq\sqsubseteq \rho'$.

Proving this theorem is more difficult than soundness. We actually have to generalize the statement of Theorem 4.4, using the predicative restriction of the $\vdash^{\mathsf{F}}$ relation, given below:

$$\dfrac{\overline{b} \# ftv(\forall \overline{a}.\rho)}{\vdash^{\mathsf{DM}} \forall \overline{a}.\rho \le \forall \overline{b}.\, [\overline{a \mapsto \tau}]\rho}\ \text{SHSUBS}$$

We write $\vdash^{\mathsf{DM}} \Gamma_2 \le \Gamma_1$ if for every $(x{:}\sigma_1) \in \Gamma_1$, there exists a σ_2 such that $(x{:}\sigma_2) \in \Gamma_2$, and $\vdash^{\mathsf{DM}} \sigma_2 \le \sigma_1$. We can now state the more general completeness statement.

Lemma 4.5 *Assume that* $\Gamma_1 \vdash e : \forall \overline{a}.\rho'$. *Then, for all* Γ_2 *with* $\vdash^{\mathsf{DM}} \Gamma_2 \le \Gamma_1$ *and for all* $\overline{\sigma}$ *there exists a* ρ_0' *such that* $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho_0'$ *and* $\rho_0' \preceq\sqsubseteq \overline{[a \mapsto \boxed{\sigma}]}\rho'$.

We also state one further corollary, which is a key ingredient to showing the implementability of the syntax-directed system by a low-level algorithm (to be described in Section 5).

Corollary 4.6 (Strengthening) *If* $\Gamma_1 \vdash^{\mathsf{sd}} e : \rho_1'$ *and* $\vdash^{\mathsf{DM}} \Gamma_2 \le \Gamma_1$ *then* $\Gamma_2 \vdash^{\mathsf{sd}} e : \rho_2'$ *such that* $\rho_2' \preceq\sqsubseteq \rho_1'$.

The proof is a combination of Theorem 4.2 and Lemma 4.5.

Corollary 4.6 means that if we change the types of expressions in the environments to be the most general according to the predicative $\vdash^{\mathsf{DM}}$, typeability is not affected. This property is important for type inference completeness for the following reason: All types that are pushed in the environment are box-free and hence can only differ from each other according to the $\vdash^{\mathsf{DM}}$ relation—their polymorphic parts are determined by annotations. In fact the algorithm will choose the most general of them according to $\vdash^{\mathsf{DM}}$. Therefore, if an expression is typeable in the declarative type system with bindings in the environments that *do not have their most general types*, the above corollary shows that the expression will also be typeable if these bindings are assigned their most general types, that is, the types that the algorithm infers for them.

5. Algorithmic implementation

The syntax-directed specification of Figure 5 can be implemented by a low-level constraint-based algorithm, which resembles the algorithm of ML^{F}. A proof-of-concept implementation, as well as the description of the algorithm invariants and properties can be found at

`www.cis.upenn.edu/~dimitriv/fph/`

Like Hindley-Damas-Milner type inference (Damas and Milner 1982; Milner 1978), our algorithm creates fresh *unification variables* to instantiate polymorphic types, and to use as the argument types of abstractions. In Hindley-Damas-Milner type inference these variables are unified with other types. Hence, a Hindley-Damas-Milner type inference engine maintains a set of *equality constraints* that map each unification variable to some type, updating the constraints as type inference proceeds.

Our algorithm uses a similar structure to Hindley-Damas-Milner type inference, but maintains both equality and *instance constraints* during type inference, so we use the term *constrained variable* instead of unification variable. A constrained variable in the algorithm corresponds to a box in the high-level specification. To distinguish between constrained variables and (rigid) quantified variables, we use greek letters α, β, for the former. Therefore, the algorithm manipulates types with the following syntax:

$$\begin{array}{lcl} \tau^\star & ::= & a \mid \tau^\star \to \tau^\star \mid \alpha \\ \rho^\star & ::= & \tau^* \mid \sigma^\star \to \sigma^\star \\ \sigma^\star & ::= & \forall \overline{a}\,.\,\rho^\star \end{array}$$

The need for instance constraints can be motivated by the typing of `choose ids` from the introduction. First, since `choose` has type $\forall a\,.\,a \to a \to a$, we may instantiate the quantified variable a with a fresh constrained variable α. However, when we meet the argument `id`, it becomes unclear whether α should be equal to $\beta \to \beta$ (that would arise from instantiating the type of `id`), or $\forall b\,.\,b \to b$ (if we do not instantiate `id`). In the high-level specification we can clairvoyantly make a (potentially boxed) choice that suits us. The algorithm does not have the luxury of clairvoyance, so rather than making a choice, it must instead simply record an *instance constraint*. In this case, the instance constraint specifies that α can be *any System F instance* of $\forall b\,.\,b \to b$. To express this, at first approximation, we need constraints of the form $\alpha \geq \sigma^\star$.

However, we need to go slightly beyond this constraint form. Consider the program `f (choose id)` where `f` has type $\forall c\,.\,c \to c$. After we instantiate the quantified variable c with a fresh variable γ, we must constrain γ by the type of `choose id`, thus

$$\gamma \geq (\text{principal type of } \texttt{choose id})$$

But, the principal type of `choose id` must be a type that is quantified *and* constrained at the same time: $[\alpha \geq \forall b\,.\,b \to b] \Rightarrow \alpha \to \alpha$. Following ML^F (Le Botlan and Rémy 2003), this *scheme* captures the *set* of all types for `choose id`, such as $\forall d\,.\,(d \to d) \to (d \to d)$ or $(\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b)$. We hence extend the bounds of constrained variables to include $\gamma \geq \varsigma$, where ς is a scheme.[4]

Schemes	ς	$::=$	$[c_1, \ldots, c_n] \Rightarrow \rho^\star$
Constraints	c	$::=$	$\alpha = \sigma^\star \mid \alpha \geq \varsigma \mid \alpha \perp$
Constraint sets	C, D, E	$::=$	$\{c_1, \ldots, c_n\}\ (n \geq 0)$

The constraint $\alpha \perp$ means that α is unconstrained. Ordinary System F types can be viewed as schemes whose quantified variables are unconstrained, and hence the type $\forall b\,.\,b \to b$ can be written as $[\beta \perp] \Rightarrow \beta \to \beta$. The meaning of the constraint $\gamma \geq \varsigma$ is that γ belongs in the *set of System F types that* ς *represents*, which we write $[\![\varsigma]\!]$. For example, if $\varsigma = [\alpha \geq ([\beta \perp] \Rightarrow \beta \to \beta)] \Rightarrow (\alpha \to \alpha)$, then we have:

$$\begin{array}{rcl} (\forall b\,.\,b \to b) \to (\forall b\,.\,b \to b) & \in & [\![\varsigma]\!] \\ \forall c\,.\,(c \to c) \to c \to c & \in & [\![\varsigma]\!] \\ \forall c\,.\,([c] \to [c]) \to [c] \to [c] & \in & [\![\varsigma]\!] \end{array}$$

[4] The actual form of constraints is slightly more complicated because we have to ensure that variables entering the environment are never equated to types with quantifiers, but we do not present it here for brevity of the exposition.

5.1 Inference implementation

The function *infer* implements our type inference algorithm, following the syntax-directed presentation of Figure 5. This function has the following signature

$$\mathit{infer} : \mathit{Constraint} * \mathit{Env} * \mathit{Term} \to \mathit{Constraint} * \mathit{Type}$$

accepting a constraint C_1, an environment Γ, and a term e. A call to $\mathit{infer}(C_1, \Gamma, e)$ either fails with *fail* or returns an updated constraint C_2 and a type $\rho^\star$. The most interesting case, which demonstrates the power of schemes, is in the implementation of applications:

$$\begin{array}{lcll} \mathit{infer}(C, \Gamma, e_1\ e_2) & = & E_1, \rho_1^\star = \mathit{infer}(C, \Gamma, e_1) & (\mathbf{1}) \\ & & E_2, \sigma_1^\star \to \sigma_2^\star = \mathit{instFun}(E_1, \Gamma, \rho_1^\star) & (\mathbf{2}) \\ & & E_3, \rho_3^\star = \mathit{infer}(E_2, \Gamma, e_2) & (\mathbf{3}) \\ & & E_4, \varsigma_3 = \mathit{generalize}(\Gamma, E_3, \rho_3^\star) & (\mathbf{4}) \\ & & E_5 = \mathit{subsCheck}(E_4, \varsigma_3, \sigma_1^\star) & (\mathbf{5}) \\ & & \mathit{inst}(E_5, \sigma_2^\star) & (\mathbf{6}) \end{array}$$

In a call to $\mathit{infer}(C, \Gamma, e_1\ e_2)$ we peform the following steps:

(1) We first infer a type $\rho_1^\star$ for e_1 and an updated constraint E_1 by calling $\mathit{infer}(C, \Gamma, e_1)$.

(2) However, type $\rho_1^\star$ may itself be a constrained type variable, that is, it may correspond to a single box in the syntax-directed specification. The function $\mathit{instFun}(E_1, \Gamma, \rho_1^\star)$ is the low-level implementation of the relation $\preceq\!\sqsubseteq^{\to}$.

(3) Subsequently, we infer a type and an updated constraint for the argument e_2 with $E_3, \rho_3^\star = \mathit{infer}(E_2, \Gamma, e_2)$.

(4) At this point we need to compare the function argument type $\sigma_1^\star$ to the type that we have inferred for the argument. However, we do not yet know the precise type of the argument and hence we call $\mathit{generalize}(\Gamma, E_3, \rho_3^\star)$ to get back a new constraint E_4 and a scheme ς_3. The scheme ς_3 expresses the set of *all possible types* of the argument e_2. The function $\mathit{generalize}$ is an appropriate generalization of the ordinary generalization of Hindley-Damas-Milner, adapted to our setting.

(5) Now that we have a scheme ς_3 expressing all possible types of the argument e_2 we must check that the required type $\sigma_1^\star$ belongs in the set that ς_3 expresses. This is achieved with the call to $\mathit{subsCheck}(E_4, \varsigma_3, \sigma_1^\star)$, which simply returns an updated constraint E_5 when $\sigma_1^\star$ belongs in the set that ς_3 denotes under the constraint E_5.

(6) Finally, type σ_2 may be equal to some type $\forall \overline{a}\,.\,\rho_2^\star$, which we instantiate to $[\overline{a \mapsto \alpha}]\rho_2^\star$ for fresh $\overline{\alpha}$. This is achieved with the call to $\mathit{inst}(E_5, \sigma_2^\star)$, which implements the $\vdash^{\mathsf{inst}}$ judgement of the syntax-directed presentation.

One may observe that in an application $e_1\ e_2$, the argument e_2 is generalized. Actually, this step is not required in a bidirectional implementation where the argument e_2 can be *checked* against the expected type that e_1 requires. We return to this point in Section 6.1.

Notice, too, that schemes make a local appearance in inference: $\mathit{generalize}$ computes a scheme (4), while $\mathit{subsCheck}$ consumes it (5). So our algorithm uses ML^F types (which is what our schemes are) *internally*, but never exposes them to the programmer. Section 7 gives some more details about the correspondence.

Another notable part of the function *infer* is related to forcing monomorphism of the constrained variables of `let`-bound expressions, or λ-abstraction bodies. Intuitively, after inferring a type for a `let`-bound expression we need to implement the instantiation along $\preceq\!\sqsubseteq$ to a box-free type. This requires that all flexible bounds inside the type of the `let`-bound expression be instantiated, and a check is performed that all constrained variables of that type are indeed mapped to monomorphic types. Then, generalization can proceed just as in ordinary Hindley-Damas-Milner.

5.2 Properties of the algorithmic implementation

We have shown termination, soundness, and completeness of the algorithmic implementation (Vytiniotis 2008). To state the soundness and completeness properties we define *boxy substitution*. Given a substitution θ from constrained variables to constrained-variable-free types, we define the boxy substitution of θ on $\sigma^\star$, denoted with $\theta[\sigma^\star]$, that substitutes the range of θ in a boxed and capture-avoiding fashion inside $\sigma^\star$. For example, if θ is the substitution $[\alpha \mapsto \sigma]$ then $\theta[\alpha \to \alpha] = \boxed{\sigma} \to \boxed{\sigma}$. We use boxy substitutions to recover specification types from algorithmic types, provided that all their constrained variables appear in the domains of the substitutions. We write $\theta\sigma^\star$ for the ordinary substitution of θ on $\sigma^\star$.

We also must connect substitutions and constraints. A substitution θ from constrained variables to System F types is said to *satisfy* a constraint C whenever it respects the interpretation of schemes in C. In particular if $(\alpha = \sigma^\star) \in C$ then it must be that $\theta\alpha = \theta\sigma^\star$ and whenever $(\alpha \geq \varsigma) \in C$ then it must be that $\theta\alpha \in [\![\theta\varsigma]\!]$.

With these two definitions, we may state soundness and completeness, connecting the syntax-directed specification with the algorithmic implementation.

Proposition 5.1 (Inference soundness) *If* $\mathit{infer}(\emptyset, \cdot, e) = C, \rho^\star$ *then for all* θ *that satisfy* C *it is the case that* $\cdot \vdash^{\mathsf{sd}} e : \theta[\rho^\star]$.

Proposition 5.2 (Inference completeness) *If* $\cdot \vdash^{\mathsf{sd}} e : \rho'$ *then* $\mathit{infer}(\emptyset, \cdot, e) = C, \rho^\star$ *and there exists a substitution* θ *that satisfies* C *and such that* $\theta[\rho^\star] \preceq \sqsubseteq \rho'$.

Together Propositions 5.1 and 5.2 ensure the implementability of the declarative specification of Figure 2. Notice that, although programs do not in general have principal types modulo boxes, a corollary of soundness and completeness is that programs with box-free types do have principal box-free types.

6. Discussion

We present in this section several extensions to FPH, and discuss alternative designs. Vytiniotis (2008) presents more details and some additional design choices.

6.1 Bidirectionality

Bidirectional propagation of type annotations may further reduce the amount of required type annotations in FPH. It is relatively straightforward to add bidirectional annotation propagation to the specification of FPH (see e.g. (Peyton Jones et al. 2007)). This bidirectional annotation procedure can be implemented as a separate preprocessing pass, provided that we support open type annotations, and annotated λ-abstractions. Alternatively, this procedure can be implemented by weaving an inference judgement of the form $\Gamma \vdash^{\mathsf{sd}} e :^{\Uparrow} \rho'$ and a checking judgement $\Gamma \vdash^{\mathsf{sd}} e :^{\Downarrow} \rho'$. Necessarily, this bidirectional system is syntax-directed. A special-top level judgement $\Gamma \vdash^{\mathsf{sd}}_\star e :^{\Downarrow} \sigma'$ checks an expression *against a polymorphic type* as follows:

$$\frac{\Gamma \vdash^{\mathsf{sd}} e :^{\Downarrow} \rho' \quad \overline{a} \# \Gamma}{\Gamma \vdash^{\mathsf{sd}}_\star e :^{\Downarrow} \forall \overline{a}\,.\,\rho'}\ \text{SKOL} \qquad \frac{\Gamma \vdash^{\mathsf{sd}} e :^{\Uparrow} \rho' \quad \overline{a} \# \Gamma \quad \vdash^{\mathsf{F}} \lfloor \forall \overline{a}\,.\,\rho' \rfloor \leq \sigma}{\Gamma \vdash^{\mathsf{sd}}_\star e :^{\Downarrow} \boxed{\sigma}}\ \text{CBOX}$$

Rule SKOL simply removes the top-level quantifiers and checks the expression against the body of the type. Rule CBOX checks an expression against a single box. In this case, we must infer a type for the expression, as we cannot use its contents.

Annotations no longer reveal polymorphism locally, but rather propagate the annotation down the term structure. The rule ANN-INF below infers a type for an annotated expression $e::\sigma$ by first *checking* e against the annotation σ:

$$\frac{\Gamma \vdash^{\mathsf{sd}}_\star e :^{\Downarrow} \sigma \quad \vdash^{\mathsf{inst}} \sigma \leq \rho'}{\Gamma \vdash^{\mathsf{sd}} (e::\sigma) :^{\Uparrow} \rho'}\ \text{ANN-INF}$$

The rule for inferring types for λ-abstractions is similar to rule SDABS, but the rule for *checking* λ-abstractions allows us now to check a function against a type of the form $\sigma'_1 \to \sigma'_2$:

$$\frac{\sigma'_1 \sqsubseteq \sigma_1 \quad \Gamma, (x{:}\sigma_1) \vdash^{\mathsf{sd}}_\star e :^{\Downarrow} \sigma'_2}{\Gamma \vdash^{\mathsf{sd}} \lambda x\,.\,e :^{\Downarrow} \sigma'_1 \to \sigma'_2}\ \text{ABS-CHECK}$$

Notice that σ'_1 must be made box-free before entering the environment, to preserve our invariant that environments are box-free.

With these additions, and assuming support for open type annotations, we can type functions with more elaborate types than simply $\tau \to \rho$ types, as the FPH original system does. Recall, for instance, Example 3.5 from Section 3.4.

```
f   :: forall a. a -> [a] -> Int
foo :: [Int -> forall b.b->b]

bog = f (\x y ->y) foo
```

Though `bog` is untypeable (even in a bidirectional system), we can recover it with the (ordinary) annotation:

```
bog = f (\x y -> y :: Int -> forall b. b -> b) foo
```

Special forms for annotated λ-abstractions (Section 3.4) are not necessary in a bidirectional system. Indeed our implementation is a bidirectional version of our basic syntax-directed type system.

6.2 η-conversion and deep instance relations

The FPH system is not stable under η-expansions, contrary to System F and ML^{F}. In particular, if $f{:}\sigma \to \mathtt{Int}$ in the environment, it is not necessarily the case that $\lambda x\,.\,f\ x$ is typeable, since x can only be assigned a τ-type.

Unsurprisingly, since FPH is based on System F, it is not stable under η-reductions. If an expression $\lambda x\,.\,e\ x$ makes a context $\mathcal{C}[(\lambda x\,.\,e\ x)]$ typeable, then it is not necessarily the case that $\mathcal{C}[e]$ is typeable. Consider the code below:

```
f   :: Int -> forall a. a -> a
g   :: forall a. a -> [a] -> a
lst :: [forall a. Int -> a -> a]

g1 = g (\x -> f x) lst   -- OK
g2 = g f lst             -- fail!
```

The application in `g2` (untypeable in implicitly typed System F) fails since `lst` requires the instantiation of `g` with type $\forall a\,.\,\mathtt{Int} \to a \to a$, whereas `f` has type $\mathtt{Int} \to \forall a\,.\,a \to a$. The FPH system, which is based on System F, is not powerful enough to understand that these two types are isomorphic.

Although such conversions are easier to support in predicative variants of Mitchell's F_η (Mitchell 1988) (e.g. (Peyton Jones et al. 2007)), the presence of impredicativity complicates our ability to support them. In fact, no type inference system with impredicative instantiations proposed to date preserves program typeability under

all η-conversions. We are currently seeking ways to extend our instance relation to some "deep" version that treats quantifiers to the right of arrows as if they were top-level, but combining that with impredicative instantiations remains a subject of future work.

6.3 Alternative design choices

Our design choices are a compromise between simplicity and expressiveness. In this section, we briefly present two alternatives.

Typing abstractions with more expressive types Recall that λ-abstractions are typed with box-free types only. This implies that certain transformations, such as *thunking*, may break typeability. For example, consider the following code:

```
f1 :: forall a. (a -> a) -> [a] -> Int
g1 = f (choose id) ids   -- OK

f2 :: forall a b. (b -> a -> a) -> [a] -> Int
g2 = f (\ _ -> choose id) ids -- fails!
```

In the example, while `g1` type checks, thunking breaks typeability, because the type $\boxed{\forall a\,.\,a \to a} \to \boxed{\forall a\,.\,a \to a}$ cannot be unboxed.

An obvious alternative would be to allow arbitrary ρ' types as results of λ-abstractions, and lift our invariant that environments are box-free to allow τ' types as the arguments of abstractions. Though such a modification allows for even fewer type annotations (the bodies of abstractions could use impredicative instantiations and no annotations would be necessary), we are not aware of a sound and complete algorithm that could implement it. Vytiniotis (2008) gives a more detailed account of the complications.

A box-free specification A safe approximation of where type annotations are necessary is at `let`-bindings or λ-abstractions with rich types. Perhaps surprisingly, taking this guideline one step further, if we *always* require annotations in bindings with rich types then we no longer need boxes in the specification *at all*! Consider the type system of Figure 2 with the following modifications:

1. Drop all boxy structure from all typing rules, that is, replace all ρ', σ', types with ρ and σ types, and completely remove SUBS and $\preceq\sqsubseteq$. Instantiate with arbitrary σ types in rule INST.
2. Replace rule LET and ABS with their corresponding versions for Damas-Milner types:

$$\frac{\Gamma \vdash u : \forall \overline{a}\,.\,\tau \qquad \Gamma, (x{:}\forall \overline{a}\,.\,\tau) \vdash e : \rho}{\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ e : \rho}\ \text{LET} \qquad \frac{\Gamma, (x{:}\tau_1) \vdash e : \tau_2}{\Gamma \vdash \lambda x\,.\,e : \tau_1 \to \tau_2}\ \text{ABS}$$

3. Add provision for annotated `let`-bindings and λ-abstractions:

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, (x{:}\sigma) \vdash e : \rho}{\Gamma \vdash \mathtt{let}\ x :: \sigma = u\ \mathtt{in}\ e : \rho}\ \text{LET-ANN}$$

$$\frac{\Gamma, (x{:}\sigma_1) \vdash e : \sigma_2}{\Gamma \vdash (\lambda x\,.\,e ::: \sigma_1 \to \sigma_2) : \sigma_1 \to \sigma_2}\ \text{ABS-ANN}$$

The resulting type system enjoys sound and complete type inference, by using essentially the same algorithm as the FPH type system. However, this variation is more demanding in type annotations than the box-based FPH. For instance, one must annotate *every* `let`-binding that uses rich types, even if its type did not involve any impredicative instantiations. For example:

```
f :: Int -> (forall a. a -> a) -> (forall a. a -> a)
h = f 42      -- fails!
```

The binding for `h` has a rich type and hence must be annotated, although no impredicative instantiation took place. Given the fact that this simplification is more demanding in type annotations, we believe that it is not really suitable for a real-world implementation.

7. Related work

There are several recent proposals for annotation-driven type inference for first-class polymorphism, which differ in simplicity of specification, implementation, placement of type annotations, and expressiveness. We present an extensive comparison below and a quick summary in Table 6.

ML^F, *Rigid* ML^F, *and HML* The ML^F language of Le Botlan and Rémy (Le Botlan and Rémy 2003; Le Botlan 2004) partly inspired this work. The biggest difference between this language and other approaches is that it extends System F types with constraints of the form $\forall(Q)\tau$ so as to recover principal types for all expressions. Therefore, `let`-expansion preserves typeability in ML^F, unlike systems that use only System F types. Because the type language is more expressive, ML^F requires strictly fewer annotations. In ML^F, annotations are necessary only when some variable is *used* at two or more polymorphic types—in contrast, in our language, variables must be annotated when they are *defined* with rich types. For example, the following program

```
f = \x -> x ids
```

needs no annotation in ML^F because x is only used once. FPH requires an annotation on `x`. Hence we are more restrictive.

A drawback of ML^F is the complexity of its specification: constrained types appear in the declarative type system and the instance relation of ML^F must include them. The FPH specification does not need a constraint-based instance relation, but our low-level implementation is a variation of the ML^F implementation. Because we do not expose a constraint-based instance relation in the specification, we can formalize our algorithm as directly manipulating sets of System F types. In contrast, ML^F internalizes the subset relation between sets of System F types as a syntactic instance relation, and formalizes type inference with respect to this somewhat complex relation. Le Botlan and Rémy (2007) study the set-based interpretation of ML^F in a recent report, which inspired our set-theoretic interpretation of schemes.

There are technical parallels between FPH and ML^F. One of the key ideas behind ML^F is that all polymorphic instantiations are "hidden" behind constrained type variables. Our type system uses anonymous boxes for the same purpose. The anonymous boxes of FPH correspond to *rigidly constrained* ML^F variables. In fact, the FPH type system can be described as a variation of ML^F without flexible bounds (Vytiniotis 2008). Additionally, usages of boxy instantiation $\preceq$ and protected unboxing $\sqsubseteq$ in an FPH typing derivation correspond to usages of the ML^F equivalences $\forall(\alpha = \sigma)\,.\,\alpha \equiv \sigma$ and $\forall(\alpha = \tau)\,.\,\sigma \equiv [\alpha \mapsto \tau]\sigma$ respectively in an ML^F derivation for the same program.

Finally, ML^F is a source language and is translated to an explicitly typed intermediate language, such as explicitly typed System F, using coercion terms (Leijen and Löh 2005). Coming up with a typed intermediate language for ML^F that is suitable for a compiler and does not require term-level coercions is still a subject of research. In contrast, because FPH is based on System F, elaborating FPH to System F is straightforward.

A variation of ML^F similar in expressive power to FPH is Leijen's Rigid ML^F (Leijen 2007). Like FPH, Rigid ML^F does not include constrained types. Instead, it resolves constraints by instantiating flexible bounds at `let`-nodes. However Rigid ML^F is specified using the ML^F instance relation. Consequently, despite the fact that types in the environment are System F types, to reason about typeability one must reason using the ML^F machinery. Additionally, the rules of Rigid ML^F require that when instantiating the types of

	Specification	Implementation	Placement of annotations / typeable programs
HM^F	Simple, "minimality" restrictions	Simple	Annotations may be needed on λ-abstractions with rich types and on arguments that must be kept polymorphic
ML^F	Heavyweight, declarative	Heavyweight	Precise, annotations only required for usage of arguments at two or more types
Boxy Types	Complex, syntax-directed, dark corners	Simple	No clear guidelines, not clear what fragment of System F is typed without annotations
HML	Constraint-based, declarative	Heavyweight	Precise, annotations on polymorphic function arguments
FPH	Simple, declarative	Heavyweight	Precise, annotations on `let`-bindings and λ-abstractions with rich types, types all applicative System F terms and more without annotations

Figure 6: Quick summary of most relevant related works

`let`-bound expressions, the type that is used in the typing derivation of the `let`-bound expression is the most general. Requiring programmers to think in terms of most general ML^F constraint-based types may even be more complicated than requiring them to reason with ML^F constraints, as in the original ML^F proposal.

A promising ML^F variation is Leijen's HML system (Leijen 2008b). In particular HML retains flexible bounds and hence enjoys principal types as ML^F, but completely inlines rigid bounds. In contrast to ML^F, annotations must be placed on *all* function arguments that are polymorphic (as in FPH), but it requires no annotations on `let`-bound definitions (contrary to FPH). The HML system still involves reasoning with constraints, but in the absence of rigid bounds there is no need for the introduction of the ML^F abstraction relation—a significant simplification.

Boxy Types Boxy Types (Vytiniotis et al. 2006) is an earlier proposal by the authors to address type inference for first-class polymorphism. Like this paper, Boxy Types uses boxed System F types to hide polymorphism. Because boxes provide an elegant way to mark impredicativity, we have reused that syntax in this work.

However, boxes play a different role in our previous work. In Boxy Types, boxes merely distinguish the parts of types that were inferred from those that result from some type annotation, combining bidirectional annotation propagation with type inference. In a Boxy Types judgement of the form $\Gamma \vdash e : \rho'$, the ρ' type should be viewed as input to the type-checker, which asks for the boxes of ρ' to get filled in. In this work, the ρ' type is an output, and boxes simply mark where impredicative instantiations took place.

Boxy Types were implemented using a relatively simple algorithm which modestly extends Hindley-Damas-Milner unification with local annotation propagation. Because the algorithm does not manipulate instance constraints, it cannot delay instantiations. Therefore, the type system must make local decisions. In particular, Boxy Types often requires programs to unbox the contents of the boxes too early. For type inference completeness, if information about the contents of a box is not locally available, it must contain a monomorphic type. As a result, the basic Boxy Types system requires many type annotations. Ad-hoc heuristics, such as N-ary applications, and elaborate type subsumption procedures, relieve the annotation burden but further complicate the specification and the predictability of the system.

Although some programs are typeable with Boxy Types and are not typeable (without annotation) in FPH, and vice versa, we believe that the simpler specification of FPH is a dramatic improvement.

HM^F Leijen's HM^F system (Leijen 2008a), which is a companion paper in this proceedings, is yet another interesting point in the design space. The HM^F system enjoys a particularly simple inference algorithm (a variant of Algorithm W), and one that is certainly simpler than FPH. In exchange, the typing rules are somewhat unconventional in form, and it is somewhat harder to predict exactly where a type annotation is required and where none is needed.

The key feature of HM^F is a clever application rule, where impredicative instantiations are determined by a local match procedure. In the type system, this approach imposes certain "minimality" conditions that require (i) that all types entering the environment are the most general types that can be assigned to programs, and (ii) that all allowed impredicative instantiations of functions are those that "minimize" the polymorphism of the returned application.

The local match procedure means that HM^F takes eager decisions: in general, polymorphic functions get instantiated by default, unless specified otherwise by the programmer. For example, the program `single id` (where `single` has type $\forall a\,.\,a \to [a]$) cannot be typed with type $[\forall a\,.\,a \to a]$. The top-level quantifiers of `id` are instantiated too early, before the local match procedure. Because FPH delays instantiations using constraints, we may type this expression with $[\boxed{\forall a\,.\,a \to a}]$ (but we would still need an annotation to `let`-bind it). In HM^F one may annotate the function `single`, or specify with a *rigid type annotation* that the type of `id` must not be instantiated: `(single (id :: forall a. a -> a))`.[5] Note that HM^F annotations are different than the annotations found, for instance, in Haskell—e.g. `(id :: forall a. a -> a) 42` is rejected.

Leijen observes that local match procedures are, in general, not robust to program transformations. If only a local match were to be used, the application `(cons id) ids` would not typeable, while `(revcons ids) id` would be (where `revcons` has type $\forall a\,.[a] \to a \to [a]$). Hence, these problems are circumvented in HM^F by using an N-ary application typing rule that uses type information from *all arguments* in an application.

In general, annotations are needed in HM^F on λ-abstractions with rich types and on arguments that must be kept polymorphic. For example, if $f : \forall a\,.\,a \to \ldots$ and $arg : \forall b\,.\,\tau$, an annotation will be needed, $f\ (arg :: \forall b\,.\,\tau)$, to instantiate a with $\forall b\,.\,\tau$. However in some cases, annotation propagation and N-ary applications may make such annotations redundant.

Because HM^F requires most general types in derivations, there are programs typeable in HM^F but not in FPH. For example, `let g = append ids in ...` requires an annotation in FPH, whereas it seamlessly typechecks in HM^F. On the other hand, flexible instantiation allows FPH to type examples such as

[5] A final possibility would be for the annotation $\forall a\,.\,a \to a$ to have been somehow *propagated* to `id`.

```
f :: forall a. [a] -> [a] -> a
g = f (single id) ids
```

where HM^F (even with annotation propagation) fails. Overall, we believe that the placement of required annotations in FPH is somewhat easier to describe than in HM^F. But on the other hand, HM^F posseses a significantly simpler implementation and metatheory.

Other works For completeness, we outline some more distantly connected works. Full type reconstruction for (implicitly typed) System F is undecidable (Wells 1999). Kfoury and Wells stratify System F types by rank (polymorhism on the left of function types), and show undecidability of type reconstruction for System F with types of rank-3 or higher. On the other hand, the rank-2 fragment of System F is decidable (Kfoury and Wells 1994).

Pfenning (1988) shows that even *partial type inference* for the n-th order polymorphic λ-calculus, where type abstractions and the positions of type applications are known but not the types of function arguments, is equivalent to n-th order unification, which is undecidable. Recent work (Le Botlan and Rémy 2007) shows that one only needs polymorphic function argument annotations (and not type abstractions and type applications) to embed all of System F. On the other hand, there are certain merits in Pfenning's proposal: (i) the type inference algorithm seamlessly extends to F_ω, (ii) higher-order unification terminates in the common case, and (iii) annotating type abstractions may be well-suited for languages with effects. On the other hand, Pfenning's original implementation effectively treats `let`-bound definitions as inlined in the body of the definition, which threatens modularity of type inference.

A different line of work explores type inference for predicative higher-rank polymorphism (Odersky and Läufer 1996; Peyton Jones et al. 2007; Rémy 2005). Odersky and Läufer made the observation that once all polymorphic function arguments are annotated, type inference for predicative higer-rank polymorphism becomes decidable, even in the presence of `let`-bound expressions. Peyton Jones *et al.* explore variations of the Odersky-Läufer type system that support a bidirectional propagation of type annotations. Finally Rémy proposed a clean separation of the bidirectional propagation of type annotations, through a phase called *shape inference*, performed before type inference.

7.1 Future work and conclusions

We have presented a simple, expressive declarative specification for type inference for impredicative polymorphism. We have implemented the system in prototype form; next, we plan to retro-fit the implementation to a full-scale compiler. We intend to address the issue of precise and informative type error reporting, a non-trivial problem as the algorithmic types are different from those of the specification. We also plan to study the interaction with type class constraints. Preliminary work by Leijen and Löh (2005) shows how to combine ML^F-style unification with qualified types, and we expect no significant difficulties to arise.

Acknowledgments The authors would like to thank Didier Rémy, Daan Leijen, the ICFP 2008 reviewers, and the Penn PLClub for many useful suggestions. This work was partially supported by NSF grants 0347289, 0702545, 0716469, and DARPA CSSG2.

References

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.

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

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.

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.

Didier Le Botlan. *MLF : Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, Ecole Polytechnique, May 2004. 326 pages, also available in english.

Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007.

Daan Leijen. HMF: simple type inference for first-class polymorphism. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*. ACM, 2008a.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. Technical Report MSR-TR-2008-55, Microsoft Research, March 2008b.

Daan Leijen. A type directed translation of MLF to System-F. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, Freiburg, Germany, 2007. ACM.

Daan Leijen and Andres Löh. Qualified types for MLF. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, pages 144–155. ACM Press, 2005.

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

John C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, 1988. ISSN 0890-5401.

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.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007. ISSN 0956-7968.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X.

Didier Rémy. Simple, partial type inference for System F, based on type containment. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 130–143, Tallinn, Estonia, September 2005. ACM.

Dimitrios Vytiniotis. *Practical type inference for first-class polymorphism*. PhD thesis, University of Pennsylvania, 2008. URL `www.cis.upenn.edu/~dimitriv/fph`. In submission.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, Portland, Oregon, 2006. ACM Press.

JB Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98:111–156, 1999.