Guarded impredicative polymorphism

Anonymous Author(s)

Abstract

The design space for type systems that support impredicative instantiation is extremely complicated. One needs to strike a balance between expressiveness, simplicity for both the end programmer and the type system implementor, and how easily the system can be integrated with other advanced type system concepts. In this paper, we propose a new point in the design space, which we call guarded impredicativity. Its key idea is that impredicative instantiation in an application is allowed for type variables that occur under a type constructor. The resulting type system has a clean declarative specification — making it easy for programmers to predict what will type and what will not —, allows for a smooth integration with GHC's OutsideIn(X) constraint solving framework, while giving up very little in terms of expressiveness compared to systems like HMF, HML, FPH and MLF. We give a sound and complete inference algorithm, and prove a principal type property for our system.

1 Introduction

Type inference for impredicative polymorphism is a deep, deep swamp. There is a dense literature of papers presenting type systems for impredicative polymorphism [1, 8, 9, 22, 23]. Alas none of them quite worked well enough to be deployed in a production compiler: either the system was too complicated for users to predict its behaviour, or it was too complicated to implement, or sometimes both.

Yet it is tantalising: what is wrong with the type $[\forall a.\, a \to a]$, a list of polymorphic functions? If we have $xs :: [\forall a.\, a \to a]$, why can't we write $(head\ xs)$? Not every programmer wants such types but, when they do, it is very annoying that they are disallowed, apparently for obscure technical reasons. That is why we keep trying.

So what is the problem? The difficulty is that to accept $(head\ xs)$ we must instantiate the type variable of *head*'s type with a polymorphic type. More precisely, since $head :: \forall p.\, [p] \to p$, we must instantiate p with $(\forall a.\, a \to a)$. This instantiation seems deceptively simple, but in practice it is extremely hard to combine with type inference. We respond to this challenge by making the following contributions:

- Every attempt to combine type inference with impredicativity involves a design trade-off between complexity, expressiveness, and annotation burden. Our key contribution is a new trade-off, which we call *guarded instantiation* or GI (Section 2).
  GI is simple: simple for the programmer to understand (Section 2.1-2.3), simple in its declarative specification and metatheory (Section 3); and simple in its implementation (Section 4). We do not extend the syntax of (System F) types in order to provide a specification of the type system (unlike previous work [1, 9, 23]), nor do we introduce new forms of annotations [19] or side-conditions that require principal types [8].
- Despite GI's relative simplicity, it accepts without annotation particularly celebrated and practically-important examples, such as *runST* \$ *e* (Figure 2).
- We give a declarative type system for GI for a small core language, highlighting the key ideas of our system (Section 3). Then we show simple extensions to handle a more full-fledged language, including type annotations (Section 3.3), **let** bindings (Section 3.4), and pattern matching (Appendix A). The system has a notion of *principal type* akin to Hindley-Milner type systems, that is, existence of a monomorphic substitution mediating between types. In particular, impredicativity is never guessed in GI (Section 3.5). The resulting system can express any System F program.
- We provide a sound and complete *inference algorithm* (Section 4) for GI, based on *constraints*. The type inference algorithm is a modest extension of the constraint-based algorithm already used by GHC. Type-correct programs can readily be elaborated into System FC, GHC's intermediate language, without extensions. Our inference algorithm scales readily to handle GADTs, type classes, higher kinds, type-level functions and other type system features.
  Moreover, as we discuss in Section 5, we can reduce the annotation burden by relaxing the guardedness conditions during solving.
- We provide a prototype implementation of the whole system, integrated with Haskell's type classes.

Type inference for impredicativity is dense with related work, as we discuss in Section 6. A small but useful contribution to a dense field is Figure 2, which presents key examples from the literature and shows how each major system behaves on that example.

2 The key idea: intuition and examples

We begin with an informal introduction to GI, which we make fully precise in Section 3. In this discussion we make use of functions defined in Figure 1.

PLDI'18, June 18–22, 2018, Philadelphia, Pennsylvania, United States
2018.

2.1 Exploiting the easy case

What is hard about typing (*head ids*)? Nothing! Since the type variable p appears under a list type constructor in *head*'s type, and we know the type of *ids*, it is plain as a pikestaff that we must instantiate $p := \forall\, a.\, a \to a$. The difficulty comes when we have a "naked" or "un-guarded" type variable in the function type, such as $single :: \forall\ p.\, p \to [p]$. Now if we examine (*single id*), it is not clear whether we should instantiate p with $\forall\, a.\, a \to a$, or with $Int \to Int$, or some other monomorphic type.

In fact, (*single id*) does not have a most general type. It has both of these two incomparable types $\forall\, a.\, [a \to a]$ and $[\forall\, a.\, a \to a]$. To make things worse (*single id*) is a perfectly typeable Hindley-Milner program (with the former type) so we must allow this type. But to support impredicativity, we must also allow the latter. But under which conditions?

Our approach is to exploit the common case. We focus on n-ary applications ($f\, e_1 \ldots e_n$). It is more conventional to deal with binary applications, but in fact n-ary applications (unencumbered with intervening **let** or **case** constructs) are wildly dominant in practice, and we can get much better typing by treating the application all at once. Then we adopt this rule to type such n-ary applications:

> The **Instantiation Rule**. When instantiating an n-ary call of $f :: \forall\, a_1 \ldots a_p.\, \tau$, a type variable a_i may only be assigned a polymorphic type σ if, *in the instantiated type*, σ appears under a type constructor in argument position $1 \ldots n$. In this case we say that a appears *guarded*.

This rule is carefully crafted. To illustrate, consider these examples (consult Figure 1 for the types):

- (*map poly*) is OK because, after instantiation, *map*'s type becomes $((\forall\, a.\, a \to a) \to (Int, Bool)) \to [\forall\, a.\, a \to a] \to [(Int, Bool)]$. The instantiating type $(\forall\, a.\, a \to a)$ appears under a type constructor of one of the supplied arguments (*poly* in this case).
- (*single ids*) is OK because, after instantiation, *single*'s type becomes $[\forall\, a.\, a \to a] \to [[\forall\, a.\, a \to a]]$. The instantiating type appears under a list type constructor in one of the supplied arguments, namely *ids*.
- ((:) *id*): here we cannot instantiate (:) at the polytype $\sigma_{id} = \forall\, a.\, a \to a$ because, because there is only one supplied argument, and σ_{id} does not appear guarded in the first argument of the instantiated type of (:). On the other hand ((:) *id ids*) would be fine.
- (*id poly* $(\lambda x.\, x)$) is a tricky one. Here *id* is applied to two arguments although its type, $\forall\, a.\, a \to a$, apparently only has one; moreover the type of *id*'s second argument must be polymorphic, and the $(\lambda x.\, x)$ must be generalised. But the Instantiation Rule says that this application is OK: in the instantiated type of *id*, the $\forall$ appears under an arrow type, in the type of the first argument.

The Instantiation Rule is still informal (which we remedy in Section 3) but it is very helpful to have a rule of thumb to explain to a programmer what will and will not work.

2.2 Ignoring the context of a call

Notice that the Intantiation Rule *takes no account of the context of the call*. For example, consider *ids* ++ *single id*. Here we append two lists, so we know that the result of (*single id*) must be $[\forall\, a.\, a \to a]$, and you might think that would be enough to fix the instantiation of *single*. But not in GI! The swamp beckons, and we stay on dry land.

Moreover, the Instantiation Rule allows the programmer to understand impredicativity in a simple bottom-up way. For example, consider the expression (*map head* (*single ids*)) (Figure 2). In our type system, the types for *head* and *single ids* are instantiated independently, so we never need to consider the *interaction* between the two arguments. This modularity pays off in the metatheory too.

There is a price to pay, however. As a degenerate case, a function application without any arguments – that is, a variable – may only instantiate *fully* monomorphically – no polymorphism, even if it appears under a type constructor. Thus, the empty list constructor $[\,] :: \forall\, a.\, [a]$ cannot be assigned a type $[\forall\, a.\, a \to a]$. We explore how to allow more programs in Section 5, and we also introduce annotations to cover the remaining cases.

2.3 Lambdas

In common with many other approaches to impredicativity, we take a conservative position on lambda-bound variables. Consider $g\ (\lambda f.\, (f\ \text{'x'}, f\ True))$, where $g :: ((\forall\, a.\, a \to a) \to (Char, Bool)) \to Int$. Since g can only be applied to a function whose argument is itself polymorphic, you could imagine that information being propagated to f and so the program could be accepted. But, in common with every other system we know, we reject all programs that require a lambda-bound variable to be polymorphic, unless it is explicitly annotated:

> The **Lambda Rule**. *Every lambda abstraction whose argument is polymorphic must be annotated. If it is not annotated, the bound variable can only have a fully monomorphic type.*

By a "fully monomorphic type" we mean "no foralls anywhere". Nothing about guardedness here! Nevertheless, in Section 5 we explore ways to alleviate some of the burden for the programmer, for cases where the argument type, albeit polymorphic, can be inferred from its usage in the body.

While the Lambda Rule deals with the arguments to lambdas, it says nothing about the return type. To get a polymorphic return type, an annotation needs to be provided. For example, for $\lambda(x :: \forall\, a.\, a \to a).\, x\ x$, GI infers the type $(\forall\, a.\, a \to a) \to b \to b$, and not $(\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a)$. To get the latter type, we have to write $\lambda(x :: \forall\ a.\, a \to a).\, (x\ x :: \forall\, a.\, a \to a)$ instead.

2.4 Expressiveness

By treating n-ary applications as a whole, and taking guardedness from both the function type and the argument types, we can infer impredicative instantiations in many practically-useful situations. We summarise a collection of examples culled from the literature in Figure 2. This table also compares our system with others, but we defer discussion of related work to Section 6.

A celebrated example is the function ($) :: $(a \to b) \to a \to b$. Haskellers use this function all the time to remove parentheses in their code, as in ($runST$ $ **do** {...}) (the type of *runST* is given in Figure 2). This call absolutely requires impredicative instantiation of the variable a in the type of ($). It is so annoying to reject this program that GHC implements a special, built-in typing rule for f $ x. Of course, that is horribly non-modular: if the programmer re-defines another version of ($), even with the same type, some programs cease to type check. In GI both type variables appear under the $(\to)$ constructor, so impredicative instantiation is allowed.

The lack of support for impredicative types is painful. For example, consider the following from Haskell's `lens` library:

$$\textbf{type}\ \mathit{Lens}\ s\ t\ a\ b = \forall f.\ \mathit{Functor}\ f \Rightarrow (a \to f\ b) \to s \to f\ t$$

Programmers think of a lens as a first-class value, and are perplexed when they cannot put a lens into a list or other data structure. With GI, many more lens-manipulating programs become well-typed.

One might worry about the order of quantifiers. Take:

$$\begin{aligned} f\ &:: (\forall\, a\ b.\ a \to b \to b) \to \mathit{Int} \\ x\ &:: \forall\, b\ a.\ a \to b \to b \\ g\ &:: [\forall\, a\ b.\ a \to b \to b] \to \mathit{Int} \\ xs &:: [\forall\, b\ a.\ a \to b \to b] \end{aligned}$$

The application $(f\ \ x)$ is well-typed, despite the differing quantifier ordering, because we compare f's argument type and x's actual type using *subsumption*; effectively we instantiate and re-generalise. In contrast, the application $(g\ \ xs)$ is ill-typed, because under a list constructor we compare the types using *equality*. Happily, while *top-level* quantifiers (such as those for x) are invisibly inferred (with unpredictable ordering), *nested* quantifiers, such as those in g and xs's type, are never inferred but rather declared through a type signature. This makes accidental incompatibility vanishingly rare in practice, as we verify in Section 5.

3 Declarative specification

We first present a systematic description of the declarative specification of GI. We use the term declarative in the sense of not syntax-directed. After we have proven soundness and completeness for the constraint-based variant, the programmer can take this declarative specification – easier to understand but without a direct inference algorithm – as a basis to understand when and why annotations are needed.

3.1 Syntax

The syntax of the core term language, for our initial discussion, is given in Figure 3. The language has some distinctive features. First, as discussed in Section 2, we deal with n-ary applications instead of binary ones. A lone term variable is treated as nullary application. Because of the Lambda Rule (Section 2.3), we provide explicitly-annotated lambda abstractions, $\lambda(x :: \sigma).\ e$, to support lambdas whose bound variable must have a polymorphic type.

Types (Figure 3) are classified by three "sorts", $\mathfrak{u}$, $\mathfrak{t}$, and $\mathfrak{m}$. *Polymorphic* types σ, ϕ, of sort $\mathfrak{u}$, have unrestricted polymorphism. *Top-level monomorphic* types, μ, η, of sort $\mathfrak{t}$, have no polymorphism at the top level, but permit arbitrary nested polymorphic types under a type constructor. Finally, *fully-monomorphic types*, τ, of sort $\mathfrak{m}$, have no trace of polymorphism. Fully monomorphic types correspond to monotypes in the Hindley-Milner tradition. These are the only types which can be assigned to un-annotated lambda-bound variables. We extend this notion to substitutions, and sometimes speak of fully monomorphic substitution to mean that the image of the substitution contains only types of that sort.

For a substitution θ, the image of a type variable a is denoted by $\theta(a)$, and similarly for sort assignments. We denote the application of a substitution to, e.g., a type scheme, or an environment, by juxtaposition.

The definitions related to the guardedness requirement are given in Figure 4. Given an n-ary application, the judgment $\sigma \rhd^n_{\mathfrak{s}} \Delta$ classifies the free type variables of σ. Specifically, Δ maps each variable a to the sort of types that are allowed to instantiate a. We combine information from multiple occurrences with $\sqcup$, the least upper bound of the (trivial) sort lattice in Figure 3; for example if a variable appear both unrestricted and monomorphically we can treat it as unrestricted. Using the $\rhd^n_{\mathfrak{s}}$ judgment, Figure 4 also defines what it means for a substitution θ to "respect the guardedness of a type", written θ respects Δ.

$$\frac{\text{for all } a \in \mathrm{dom}(\theta), \vdash \theta(a) : \Delta(a)}{\theta \text{ respects } \Delta}$$

3.2 Typing rules

The typing judgment $\Gamma \vdash e : \sigma$ is given in Figure 5, along with some auxiliary judgments.

Rules Abs and AnnAbs concern lambda abstractions, and are straightforward. AnnAbs deals with a lambda $(\lambda(x :: \phi).\ e)$ where the user has supplied a type annotation ϕ: we simply bring x into scope with type ϕ. As discussed in Section 2.3, where there is no annotation (rule Abs) we insist that x has a fully-monomorphic type τ.

All the action is in rule App for n-ary applications $(h\ e_1 \ldots e_n)$. The first step is typing the head of the application h. The corresponding judgment $\vdash^{\mathsf{h}}$ either looks for a variable in the environment or uses the normal typing judgment if the head is another kind of term.

$head :: \forall p. [p] \to p$	$id :: \forall a. a \to a$	$ids :: [\forall a. a \to a]$
$tail :: \forall p. [p] \to [p]$	$inc :: Int \to Int$	$map :: \forall p\, q. (p \to q) \to [p] \to [q]$
$[\,] :: \forall p. [p]$	$twice :: \forall a. (a \to a) \to a \to a$	$app :: \forall a\, b. (a \to b) \to a \to b$
$(:) :: \forall p. p \to [p] \to [p]$	$choose :: \forall a. a \to a \to a$	$revapp :: \forall a\, b. a \to (a \to b) \to b$
$single :: \forall p. p \to [p]$	$poly :: (\forall a. a \to a) \to (Int, Bool)$	$flip :: \forall a\, b\, c. (a \to b \to c) \to b \to a \to c$
$(+\!\!+) :: \forall p. [p] \to [p] \to [p]$	$auto :: (\forall a. a \to a) \to (\forall a. a \to a)$	$runST :: \forall v. (\forall s. ST\ s\ v) \to v$
$length :: \forall p. [p] \to Int$	$auto' :: (\forall a. a \to a) \to b \to b$	$argST :: \forall s. ST\ s\ Int$

Figure 1. Type signatures for functions used in the text

After typing the head, we instantiate the type of the head by means of the $\leqslant_{\mathfrak{s}}^n$ judgment. Note that this *instantiation* judgment is parametrized by a sort $\mathfrak{s}$ (this is needed to support applications with and without annotations).

- All type constructors are *invariant*, even functions. This means that neither $[\forall a. a \to a] \nleqslant_{\mathfrak{s}}^n [Int \to Int]$ nor $Int \to (\forall a. a \to a) \nleqslant_{\mathfrak{s}}^n Int \to Int \to Int$.
- For function types the judgment $\leqslant_{\mathfrak{s}}^n$ embodies a bit of their covariance. Given a function f of the type $\forall\ a. a \to (\forall\ b. b \to a)$ we are allowed to first instantiate a with some type τ_1, which returns a result $\forall\ b. b \to \tau_1$, and then instantiate b if a second argument is supplied.
  As a consequence, we can work around the lack of variance of function types by η-expanding. In the previous example, if we need an expression of type $\tau_1 \to \tau_2$ for some hole, we cannot directly use f. But we can use $\lambda x\ y. f\ x\ y$ instead, which allows the $\leqslant_{\mathfrak{s}}^2$ judgment to kick in and instantiate the two type variables.
- The status of the variables in the result type depends on the parameter $\mathfrak{s}$ to the judgment. In the case of App, this parameter is set to $\mathfrak{m}$. As a consequence, those variables appearing only in the result of the function – the part of the type past the given number of arguments – have to be instantiated with fully monomorphic types.

Whereas the type of the head of the application may only be instantiated, the arguments may also ongo *generalisation*, that is, abstraction of some of the type variables. Generalisation is embodied by the ArgGen rule. Without such a rule, we would not be able to type check *twice* $(\lambda x. x)$. Here the type of $\lambda x. x$ must be of the form $\tau \to \tau$ for some fully monomorphic τ. If we choose τ to be a fresh Skolem variable, we can derive $\Gamma \vdash^{\text{arg}} \lambda x. x : \forall a. a \to a$, as needed.

Theorem 3.1 (Compatibility with Hindley-Milner)**.** *Let e be an expression in the syntax of Hindley-Milner without* **let**. *If $\Gamma \vdash^{\text{HM}} e : \tau$, then $\Gamma \vdash e : \tau$.*

3.3 Annotations in applications

With the rules in Figure 5, a single variable (not applied to any arguments) is treated as a nullary application. Rule App cannot get any guardedness information from its arguments, and its type can only be instantiated with fully monomorphic types. As a result, *we cannot assign the polymorphic type $[\forall a. a \to a]$ to the empty list* $[\,]$. That is embarrassing, because we cannot typecheck, say $([\,] +\!\!+ ids)$. Figure 2 has other examples.

One solution is to allow the programmer to give a type annotation for an application; see the syntax in Figure 6. Now, since annotations fully specify the types, we do not need to impose guardedness restrictions on those variables appearing in the result. Thus, the expression $[\,] :: [\forall a. a \to a]$ is accepted, even though the type variable in a is not guarded by a type constructor. GI is quite symmetrical in terms of syntax: both abstractions and applications may be annotated.

Annotations also free us from having a different judgment for declarations and expressions. For every combination $f :: \sigma; f = e$ in the source code, we just need to pose the problem of checking $f = e :: \sigma$ for well-typedness.

The extensions required for annotated applications are described in Figure 6. Rule AnnApp is almost identical to App, except for the choice of parameter to the instantiation judgment, which is $\mathfrak{u}$. This implies that in contrast to non-annotated applications, variables in the result type of the function might be substituted by any type, polymorphic or not. This is sensible, the annotation tells us exactly what the types are that those variables should be instantiated with.

Of course, annotated applications serve for arbitrary applications, not just nullary ones. Take the expression *single* $(\lambda x. x)$. Due to the type of *single* being $\forall p. p \to [p]$, the type of the expression must be $[\tau \to \tau]$ for a monomorphic τ. If we want instead to obtain $[\forall a. a \to a]$, we can just annotate the result, thus *single* $(\lambda x. x) :: [\forall a. a \to a]$. Since the variable a appears in the result type of *single*, this is perfectly fine by rule AnnApp.

There is some room for choice when introducing annotations in the language. In particular, the annotations can be taken as *rigid* – the type of the decorated expression is exactly the one appearing as annotation – or *soft* – the resulting type might be an instance of the one stated in the annotation. We take the former route, which is shared by most of the previous work in impredicative polymorphism.

3.4 let bindings

Following the long-established tradition, we could translate **let** $x = e_1$ **in** e_2 as $(\lambda x. e_2)\ e_1$. Alas, such a translation imposes an important restriction on the type of x: it must

	GI decl.	GI relax.	MLF	HMF	FPH	HML
POLYMORPHIC INSTANTIATION						
$const2 = \lambda x\ y.\ y$	✓	✓	✓	✓	✓	✓
MLF infers $(b \geqslant \forall c.\ c \to c) \Rightarrow a \to b$, GI infers $a \to b \to b$.						
choose id	✓	✓	✓		✓	✓
MLF and HML infer $(a \geqslant \forall b.\ b \to b) \Rightarrow a \to a$, FPH and GI infer $(a \to a) \to a \to a$.						
choose [ ] ids	No	✓	✓	✓	✓	✓
GI needs an annotation on $[\,] :: [\forall a.\ a \to a]$.						
$\lambda(x :: \forall a.\ a \to a).\ x\ x$	✓	✓	✓	No	✓	✓
MLF infers $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$, GI infers $(\forall a.\ a \to a) \to b \to b$.						
id auto	✓	✓	✓	✓	✓	✓
id auto′	✓	✓	✓	✓	✓	✓
choose id auto	No	✓	✓	No	No	✓
choose id auto′	No	No	✓	No	No	✓
f (choose id) ids	No	✓	✓	No	✓	✓
where $f :: \forall a.\ (a \to a) \to [a] \to a$						
GI needs an annotation on $id :: (\forall a.\ a \to a) \to (\forall a.\ a \to a)$ in the previous two examples.						
poly id	✓	✓	✓	✓	✓	✓
$poly\ (\lambda x.\ x)$	✓	✓	✓	✓	✓	✓
$id\ poly\ (\lambda x.\ x)$	✓	✓	✓	✓	✓	✓
INFERENCE OF POLYMORPHIC ARGUMENTS						
$\lambda f.\ (f\ 1, f\ True)$	No	No	No	No	No	No
All systems require an annotation on $f :: \forall a.\ a \to a$.						
$\lambda xs.\ poly\ (head\ xs)$	No	No	✓	No	No	No
All systems except for MLF require an annotation on $xs :: [\forall a.\ a \to a]$.						
FUNCTIONS ON POLYMORPHIC LISTS						
length ids	✓	✓	✓	✓	✓	✓
tail ids	✓	✓	✓	✓	✓	✓
head ids	✓	✓	✓	✓	✓	✓
single id	✓	✓	✓	✓	✓	✓
id : ids	✓	✓	✓	No	✓	✓
$(\lambda x.\ x) : ids$	✓	✓	✓	No	✓	✓
single inc ++ single id	✓	✓	✓	✓	✓	✓
g (single id) ids	No	No	✓	No	✓	✓
where $g :: \forall a.\ [a] \to [a] \to a$						
map poly (single id)	No	No	✓	✓	✓	✓
GI needs an annotation on $single\ id :: [\forall a.\ a \to a]$ in the previous two examples.						
map head (single ids)	✓	✓	✓	✓	✓	✓
APPLICATION FUNCTIONS						
app poly id	✓	✓	✓	✓	✓	✓
revapp poly id	✓	✓	✓	✓	✓	✓
runST argST	✓	✓	✓	✓	✓	✓
app runST argST	✓	✓	✓	✓	✓	✓
revapp runST argST	✓	✓	✓	✓	✓	✓
η-EXPANSION						
k h lst	No	No	No	No	No	No
$k\ (\lambda x.\ h\ x)\ lst$	✓	✓	✓	No	✓	✓
where $h :: Int \to \forall a.\ a \to a$, $k :: \forall a.\ a \to [a] \to a$, and $lst :: [\forall a.\ Int \to a \to a]$						

Figure 2. Comparison of type systems

Skolem / rigid variables	$\mathbb{V}$	$\ni$	$a, b, c, \ldots$
Type constructors	$\mathbb{T}$	$\ni$	$\to, \mathsf{T}, \mathsf{S}, \ldots$
Fully mono. types	τ	$::=$	$a \mid \mathsf{T}\ \overline{\tau}$
Top-level mono. types	μ, η	$::=$	$a \mid \mathsf{T}\ \overline{\sigma}$
Polymorphic types	σ, ϕ	$::=$	$\forall \overline{a}.\ \mu$
			$\overline{a}$ may be empty, $\overline{a} \subseteq \mathrm{ftv}(\mu)$
Term variables		$\ni$	x, y
Heads	h	$::=$	$x \mid e$
Expressions / terms	e	$::=$	$h\ e_1\ \ldots\ e_n$
		$\mid$	$\lambda x.e$
		$\mid$	$\lambda(x :: \sigma).e$
Environments	Γ	$::=$	$\epsilon \mid \Gamma, x : \sigma$
Substitutions	θ, φ, π	$::=$	$[\overline{a \mapsto \sigma}]$
Sorts of variables	$\mathfrak{s}$	$::=$	$\mathfrak{u} \mid \mathfrak{t} \mid \mathfrak{m}$
			Sorts form a lattice, $\mathfrak{m} \sqsubset \mathfrak{t} \sqsubset \mathfrak{u}$
Sort assignment	Δ	$::=$	$[\overline{a \mapsto \mathfrak{s}}]$
Free type variables	$\mathrm{ftv}(\sigma)$		

Figure 3. Syntax of the language

be fully monomorphic even though the type of e_1 might be more general. The reason is that we try to guess the type of e_1 by looking at the way it is used in e_2, instead of looking at e_1 itself. But there is no need to be so restrictive! The rule Let in Figure 6 works in the other direction: the type obtained from typing e_1 is put in the environment as the one for x, allowing the type of x to be fully polymorphic instead.

One difference between **let** bindings in GI and Hindley-Milner is that the latter always generalizes the type of a let-bound identifier before passing it to the body of the let. Vytiniotis et al. [21] argue however that **let**-generalisation is not so important in practice and that in complex type systems how to generalize is not completely clear. If desired, generalisation can be obtained by annotating the bound expression, **let** $x = (e_1 :: \phi)$ **in** e_2.

3.5 Metatheory

One key property of GI is that all impredicative instantiations are settled by the shape of the expression and the types in the environment. Formally, every possible type derivation for an expression e results in the same type, modulo some monomorphic substitution. For that reason, we say that impredicative polymorphism is *not guessed* in GI.

Theorem 3.2 (Impredicative instantiation is not guessed)**.** *Let Γ be a (possibly open) environment and e an expression. For every pair of fully monomorphic substitutions θ_1 and θ_2,*

1. *If $\theta_1\Gamma \vdash^{h} e : \sigma_1$ and $\theta_2\Gamma \vdash^{h} e : \sigma_2$, then there exists a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*
2. *If $\theta_1\Gamma \vdash e : \sigma_1$ and $\theta_2\Gamma \vdash e : \sigma_2$, then there exists a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i\sigma^*$.*

$$\boxed{\sigma \vdash a \text{ guarded}}$$

$$\frac{a \notin \overline{b} \qquad \mu \vdash a \text{ guarded}}{\forall \overline{b}.\, \mu \vdash a \text{ guarded}} \qquad \frac{a \in \text{ftv}(\overline{\phi})}{\mathsf{T}\,\overline{\phi} \vdash a \text{ guarded}}$$

$$\boxed{\vdash \sigma : \mathfrak{s}}$$

$$\frac{}{\vdash \sigma : \mathfrak{u}} \qquad \frac{}{\vdash \mu : \mathfrak{t}} \qquad \frac{}{\vdash \tau : \mathfrak{m}}$$

$$\boxed{\sigma \rhd^{n}_{\mathfrak{s}} \Delta}$$

$$\frac{}{\mu \rhd^{0}_{\mathfrak{s}} [\text{ftv}(\mu) \mapsto \mathfrak{s}]} \qquad \frac{\mu \rhd^{n}_{\mathfrak{s}} \Delta}{\forall \overline{a}.\, \mu \rhd^{n}_{\mathfrak{s}} \Delta \backslash \overline{a}} \qquad \frac{\sigma_2 \rhd^{n-1}_{\mathfrak{s}} \Delta \qquad \hat{\Delta} = [a \mapsto \text{ if } \sigma_1 \vdash a \text{ guarded}, \mathfrak{u} \text{ else } \mathfrak{t} \mid a \in \text{ftv}(\sigma_1)]}{\sigma_1 \to \sigma_2 \rhd^{n}_{\mathfrak{s}} \Delta \sqcup \hat{\Delta}}$$

Figure 4. Definitions related to guardedness

$$\boxed{\Gamma \vdash^{\mathsf{h}} e : \sigma} \qquad \boxed{\Gamma \vdash^{\mathsf{arg}} e : \sigma}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}} x : \sigma}\ \textsc{VarHead} \qquad \frac{e \text{ not var. or app.} \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash^{\mathsf{h}} e : \sigma}\ \textsc{ExprHead} \qquad \frac{\Gamma \vdash e : \sigma \qquad \overline{b} \text{ fresh} \qquad \sigma \leqslant^{0}_{\mathfrak{m}} [\overline{a \mapsto b}]\mu}{\Gamma \vdash^{\mathsf{arg}} e : \forall \overline{a}.\, \mu}\ \textsc{ArgGen}$$

$$\boxed{\sigma \leqslant^{n}_{\mathfrak{s}} \phi_1, \ldots, \phi_n, \mu}$$

$$\frac{}{\mu \leqslant^{0}_{\mathfrak{s}} \mu}\ \textsc{InstMono} \qquad \frac{n \geqslant 1 \qquad \phi_2 \leqslant^{n-1}_{\mathfrak{s}} \sigma_2, \ldots, \sigma_n, \mu}{\phi_1 \to \phi_2 \leqslant^{n}_{\mathfrak{s}} \phi_1, \sigma_2, \ldots, \sigma_n, \mu}\ \textsc{InstArrow} \qquad \frac{\mu \rhd^{n}_{\mathfrak{s}} \Delta \qquad \theta \text{ respects } \Delta \qquad \theta\mu \leqslant^{n}_{\mathfrak{s}} \overline{\sigma}, \eta}{\forall \overline{a}.\, \mu \leqslant^{n}_{\mathfrak{s}} \overline{\sigma}, \eta}\ \textsc{InstPoly}$$

$$\boxed{\Gamma \vdash e : \sigma}$$

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}\ \textsc{Abs} \qquad \frac{\Gamma, x : \phi \vdash e : \sigma}{\Gamma \vdash \lambda(x :: \phi).\, e : \phi \to \sigma}\ \textsc{AnnAbs}$$

$$\frac{\Gamma \vdash^{\mathsf{h}} h : \phi \qquad \phi \leqslant^{n}_{\mathfrak{m}} \sigma_1, \ldots, \sigma_n, \mu \qquad \Gamma \vdash^{\mathsf{arg}} e_1 : \sigma_1 \ldots \Gamma \vdash^{\mathsf{arg}} e_n : \sigma_n}{\Gamma \vdash h\, e_1\, \ldots\, e_n : \mu}\ \textsc{App}$$

Figure 5. Declarative type system

$$\text{Expressions / terms} \quad e ::= \ \ldots \mid (h\, e_1\, \ldots\, e_n :: \sigma) \mid \textbf{let } x = e_1 \textbf{ in } e_2$$

$$\frac{\Gamma \vdash^{\mathsf{h}} h : \phi \qquad \phi \leqslant^{n}_{\mathfrak{u}} \sigma_1, \ldots, \sigma_n, \eta \qquad \Gamma \vdash^{\mathsf{arg}} e_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash^{\mathsf{arg}} e_n : \sigma_n}{\Gamma \vdash (h\, e_1\, \ldots\, e_n :: \forall \overline{b}.\, \eta) : \forall \overline{b}.\, \eta}\ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_1 : \phi \qquad \Gamma, x : \phi \vdash e_2 : \sigma}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \sigma}\ \textsc{Let}$$

Figure 6. Decl. system with annotations in apps. and **let**

Corollary 3.3. *Let Γ be a closed environment. If $\Gamma \vdash e : \sigma_1$ and $\Gamma \vdash e : \sigma_2$, then there is a polymorphic type σ^* and fully monomorphic substitutions φ_1 and φ_2 such that $\sigma_i = \varphi_i \sigma^*$.*

This property suggests a notion of *principal type* similar to the one found in Hindley-Milner. A principal type for an expression e is defined as a type σ^* for which any other type assignment ϕ to e is equal to $\theta\sigma^*$ for a fully monomorphic substitution θ. The fact that we only need to consider *fully monomorphic* substitutions here is a direct consequence of Theorem 3.2. The proof of the principal types property, however, is a corollary of other properties of the inference process, which we describe in Section 4.4.

In the remainder of this section we look at some properties of GI concerning derivations and stability under transformations. The latter are important as they provide a basis for the compiler to optimize the code while respecting the typing semantics. Proofs are given in Appendix E.1.

Theorem 3.4. *If $\Gamma \vdash u : \sigma$ and $\Gamma, x : \sigma \vdash e[x] : \phi$, then $\Gamma \vdash e[u] : \phi$.*

Theorem 3.5. *Let $app :: \forall\ a\ b.\, (a \to b) \to a \to b$ and $revapp :: \forall\ a\ b.\, a \to (a \to b) \to b$ be the application and reverse application functions, respectively. Given two expressions*

Unif. var. names	$\mathbb{U}$	$\ni$	$\alpha, \beta, \gamma, \delta, \ldots$
Unif. var.	υ	::=	$\alpha^{\mathfrak{s}}$
Fully mono. types	τ	::=	$\alpha^{\mathfrak{m}} \mid a \mid \mathsf{T}\,\overline{\tau}$
Top-level mono. types	μ, η	::=	$\alpha^{\mathfrak{t}} \mid a \mid \mathsf{T}\,\overline{\sigma}$
Polymorphic types	σ, ϕ	::=	$\alpha^{\mathfrak{u}} \mid \forall \overline{a}.\, \mu$
	$\overline{a}$ may be empty, $\overline{a} \subseteq \mathsf{ftv}(\mu)$		
Types with generalisation	$\mathfrak{g}$	::=	$\forall \{\overline{\upsilon}\}.\, C \Rightarrow \sigma$
	$\overline{\upsilon}$ may be empty		
Constraints	C	::=	$\top$
Conjunction		$\mid$	$C_1 \wedge C_2$
Equality		$\mid$	$\sigma \sim \phi$
Instantiation		$\mid$	$\sigma \leqslant^{n}_{\mathfrak{s}} \mu$
Generalisation		$\mid$	$\mathfrak{g} \preceq \sigma$
Quantification		$\mid$	$\forall \overline{a}.\, \exists \overline{\upsilon}.\, C$
Free unification variables	$\mathsf{fuv}(\sigma)$		

Figure 7. Extended syntax

f and e such that $\Gamma \vdash^{\mathsf{h}} f : \sigma_0$, and $\sigma_0 \leqslant^{0}_{\mathfrak{m}} \sigma_1 \rightarrow \phi$ then:

$$\Gamma \vdash f\ e : \phi \iff \Gamma \vdash \mathit{app}\ f\ e : \phi \iff \Gamma \vdash \mathit{revapp}\ e\ f : \phi$$

The hypothesis $\sigma_0 \leqslant^{0}_{\mathfrak{m}} \sigma_1 \rightarrow \phi$ means that type variables may only be instantiated with fully monomorphic variables. Thus, this transformation only respects well-typedness for the whole predicative and higher-rank fragment, but not in the fully impredicative system in general.

4 Type inference using constraints

In the previous section we described GI from a declarative perspective and now we turn to describing an efficient type inference algorithm for it.

Following Pottier and Rémy [15], we first walk over the syntax tree of the source program and generate *typing constraints*, a process that typically introduces many *unification variables* that stand for as-yet-unknown types. Next, we solve those constraints producing a *type substitution* for these unification variables. By separating type inference in two simpler problems, the implementation and conceptual overhead with new source language and type system features remains low. For example, earlier work dubbed OutsideIn(X) applies these ideas to a language like Haskell, with type classes, type-level functions, GADTs, and the like [21]. Another advantage is more sophisticated type-error diagnosis [6, 20, 25].

4.1 Constraints

The main challenge of type inference for impredicativity concern instantiation and generalisation of terms with polymorphic type. Consider the call (*head ids True*), when fully elaborated we want to generate this System F term:

$$\mathit{head}\ (\forall\, a.\, a \rightarrow a)\ \mathit{ids}\ \mathit{Bool}\ \mathit{True}$$

That is, we instantiate *head* at type $(\forall\, a.\, a \rightarrow a)$, then apply it to *ids*, to produce a result of type $(\forall\, a.\, a \rightarrow a)$. Now we must in turn instantiate that type with *Bool* to get a function of type $(\mathit{Bool} \rightarrow \mathit{Bool})$ which we can apply to *True*. This second instantiation is problematic because, at constraint generation time, we do not yet know what type we are going to instantiate *head* at; all we know is that (*head ids*) has type α for some as-yet-unknown type α. So we want to defer the instantiation decision.

Sometimes we must defer generalisation decisions too. For example, consider the function application $((:)\ (\lambda x.\, x)\ \mathit{ids})$. In System F terms, we are trying to infer the following elaborated program:

$$(:)\ (\forall\, a.\, a \rightarrow a)\ (\Lambda a.\, \lambda(x : a).\, x)\ \mathit{ids}$$

in which (:) is instantiated at type $(\forall\, a.\, a \rightarrow a)$, and (:)'s first argument is generalised to have that polymorphic type. Now consider constraint generation for this expression. We may instantiate the type of (:) with a fresh unification variable, α say. Ultimately the type of *ids* forces α to be $\forall\, a.\, a \rightarrow a$, but we don't know that yet. Moreover, in the final program we will need to generalise the type of $(\lambda x.\, x)$, but again at constraint generation time we don't know that type either.

When we don't know something at constraint generation time, the solution is to *defer the choice, by generating a constraint that represents that choice.* This is the key idea of the constraint solving approach. The game is to develop a constraint language that neatly embodies the choices that we want to defer, and a solver that can subsequently make those choices. With that in mind, Figure 7 gives the syntax of our constraint language.

As mentioned earlier, constraint generation produces many *unification variables*, each of which stands for an as-yet unknown type. Looking at Figure 7, a key idea is that unification variables are drawn from three distinct "alphabets": $\alpha^{\mathfrak{s}}$ for each of the threes sorts $\mathfrak{s}$. (Sorts were introduced in Figure 3.) The sort of a unification variable specifies the possible types that the unification variable can stand for; operationally, a unification variable of sort $\mathfrak{s}$ may only be unified with types belonging to that sort.

The syntax presents several kinds of constraints C. Equality constraints are self explanatory. Instantiation constraints arise from the occurrence of a polymorphic variable, whose type must be instantiated – but that decision must be deferred (embodied in a constraint). Quantification constraints arise from explicit user type signatures, and pattern matching on data types involving existentials and GADTs. Both are fairly conventional. However *generalisation constraints* are new; they precisely embody the deferred decision about generalisation that we mention above. We will elaborate on all these forms in what follows.

In GI, constraints do not form part of the source language; they are internal to the solver. But once we extend this approach to work with type system extensions such as type classes in Appendix B, we shall impose a distinction between simple constraints – those which the programmer can type – and extended ones – which are internal.

4.2 Constraint generation

Constraint *generation* is described in Figure 8 as a four-element judgment $\Gamma \vdash e : \sigma \rightsquigarrow C$. The first two elements are inputs: the environment Γ and the expression e for which to generate constraints. The output of the process is a type σ assigned to the expression, possibly including some unification variables, and the set of extended constraints C that the types must satisfy. We first focus on the generation process for the core calculus, which we extend later to cover the rest of the language.

Rules Abs and AbsAnn are not surprising: they just extend the environment with a new unification variable or a given polymorphic type, respectively, and then proceed to generate constraints for the body of the abstraction. The usage of a fully monomorphic variable in Abs mimics the restriction imposed by the declarative specification.

Rule App is where most of the work is done. Just like the declarative specification (Figure 5), the head of the application is typed using an ancillary judgment $\Gamma \vdash^{h} h : \phi \rightsquigarrow C$, which either looks up a variable in the environment or threads the information to the normal gathering process.

The first column of constraints, of form $\beta^{u}_{i} \leqslant^{n-i}_{m} \alpha^{u}_{i+1} \to \beta^{u}_{i+1}$, successively decomposes the function type ϕ to expose its arguments. At each argument we may need to instantiate the top-level foralls of the function type to expose the arrow; hence the use of subsumption $\leqslant^{n}_{s}$ rather than type equality. Note that what we express declaratively in a single relation $\sigma_0 \leqslant^{n}_{s} \sigma_1, \ldots, \sigma_n, \mu$ is expressed as a sequence of constraints

$$\sigma_0 \leqslant^{n}_{s} \sigma_1 \to \phi_1 \qquad \phi_1 \leqslant^{n-1}_{s} \sigma_2 \to \phi_2$$
$$\ldots \qquad \phi^{n-1} \leqslant^{1}_{s} \sigma_n \to \phi_n \qquad \phi_n \leqslant^{0}_{s} \mu$$

The reason will become apparent once we describe how solving proceeds.

The second column of constraints introduces a completely new constraint form, $\mathrm{g} \preceq \sigma$, which we call *generalisation* constraint. These constraints allow us to defer the generalisation decisions to the solver, as sketched in Section 4.1. The constraint $(\forall\{\overline{v}\}. C \Rightarrow \phi) \preceq \sigma$ should be read *"a term of type ϕ with constraints C and unification variables $\overline{v}$ can be instantiated and/or generalised to have type σ"*. Even looking at the syntax alone, you can see that the fruits of constraint generation for each argument e_i are wrapped up, along with the expected argument type α_i from the function, into a generalisation constraint for the solver to deal with later.

Rule AnnApp deals with a type-annotated application $(h\, e_1\, \ldots\, e_n :: \sigma)$. It is similar to App, but it is the first rule to introduce a *quantification constraint* $\forall\overline{b}.\, \exists\overline{v}.\, \overline{C}$. This binds the Skolem variables $\overline{b}$ from the type signature, and existentially quantifies the unification variables free in the constraint but not used outside it. The other significant difference is the use of $\leqslant^{n}_{u}$ in the column of instantiation constraints, with suffix u, rather than m in rule App, exactly following the difference between AnnApp and App in Figure 6 and 5 resp.

4.3 Constraint solving

The solver takes the generated constraint C and its free unification variables $\overline{v} = \mathrm{fuv}(C)$, and repeatedly applies the solver rules in Figure 9, until no rule applies. The result is a *residual constraint*. If the residual constraint is in *solved form*, then the program is well typed; if not, the unsolved constraints (e.g. Int ~ Bool) represent type errors that can be reported to the user. We will discuss solved form shortly, in Section 4.3.3, but first we concentrate on the solver rules that incrementally solve the constraint.

Each of the rules in Figure 9 rewrites a configuration $C; \overline{v}$ to another configuration. The unification variables $\overline{v}$ are existentially quantified, so you can think of a configuration as representing $\exists\overline{v}.C$. Rule conj and forall are structural rules: the former allows a rule to be applied to one part of a conjunction, while the latter allows a rule to be applied under a quantification. To avoid clutter we implicitly assume that the rules are read modulo commutativity and associativity of $\wedge$; that is why conj only has to handle the left conjunct.

4.3.1 Basic rules

Rule eqrefl removes trivial equality constraints $\sigma \sim \sigma$. Rule eqmono indicates that two types headed by constructors are equal if and only if their heads coincide and all the arguments are equal. eqsubst is the only rule that involves the interaction of two constraints. It applies the substitution of a unification variable to any other constraints conjoined with it (remember the implicit associativity and commutativity of $\wedge$), provided sorts are respected, and there is no occurs-check. Notice that the equality constraint is not discarded; it remains in case it is needed again; indeed, these equality constraints remain in a solved constraint (Section 4.3.3).

Given this different behaviour of the different sorts of variables, the solver has to propagate this information. eqvar ensures that whenever we have two variables with different sorts, the least restrictive one is substituted by the most restrictive. For example, when we have an unrestricted α^{u} and a top-level monomorphic β^{t}, then α^{u} should be replaced by β^{t}, and not the other way around. Full monomorphism goes deeper: eqfully ensures that if a type σ is equated with a fully monomorphic variable α^{m}, all the variables in σ become fully monomorphic too.

One difference between these rules and other presentations is that we do not rewrite an unsatisfiable constraint, such as *Int* ~ *Bool*, to $\bot$. Instead, that constraint is simply stuck, and we can report it at the end.

Note that two polymorphic types need to be *syntactically equal* (modulo α-equality) to match under the eqrefl rule. This means that $(\forall\, a\, b.\, a \to b \to b) \sim (\forall\, b\, a.\, a \to b \to b)$ does *not* hold in our system. As we discuss in Section 2.4,

$$\boxed{\Gamma \vdash^{\mathsf{h}} e : \sigma \rightsquigarrow C}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}} x : \sigma \rightsquigarrow \epsilon}\ \textsc{VarHead} \qquad \frac{e \text{ not var. or app.} \qquad \Gamma \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash^{\mathsf{h}} e : \sigma \rightsquigarrow C}\ \textsc{ExprHead}$$

$$\boxed{\Gamma \vdash e : \sigma \rightsquigarrow C}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x : \alpha^{\mathfrak{m}} \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash \lambda x.\, e : \alpha^{\mathfrak{m}} \to \sigma \rightsquigarrow C}\ \textsc{Abs} \qquad \frac{\Gamma, x : \phi \vdash e : \sigma \rightsquigarrow C}{\Gamma \vdash \lambda(x :: \phi).\, e : \phi \to \sigma \rightsquigarrow C}\ \textsc{AnnAbs}$$

$$\frac{\Gamma \vdash^{\mathsf{h}} h : \phi \rightsquigarrow C \qquad \begin{array}{l}\Gamma \vdash e_i : \sigma_i \rightsquigarrow C_i \\ \overline{v_i} = \mathrm{fuv}(\sigma_i, C_i) - \mathrm{fuv}(\Gamma, \phi, C)\end{array} \qquad \overline{\alpha}, \overline{\beta}, \delta \text{ fresh}}{\begin{array}{lllll}\Gamma \vdash h\, e_1 \ldots e_n : \delta_{\mathrm{t}} \rightsquigarrow C & \wedge\ \ \phi \leqslant^{n}_{\mathfrak{m}} \alpha^{\mathfrak{u}}_1 \to \beta^{\mathfrak{u}}_1 & \wedge\ \ \forall\{\overline{v_1}\}.\, C_1 \Rightarrow \sigma_1 \leq \alpha^{\mathfrak{u}}_1 & & \\ & \wedge\ \ \beta^{\mathfrak{u}}_1 \leqslant^{n-1}_{\mathfrak{m}} \alpha^{\mathfrak{u}}_2 \to \beta^{\mathfrak{u}}_2 & \wedge\ \ \forall\{\overline{v_2}\}.\, C_2 \Rightarrow \sigma_2 \leq \alpha^{\mathfrak{u}}_2 & & \\ & \wedge\ \ \vdots & \wedge\ \ \vdots & \wedge\ \ \beta^{\mathfrak{u}}_n \leqslant^{0}_{\mathfrak{m}} \delta^{\mathrm{t}} & \end{array}}\ \textsc{App}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{\mathsf{h}} h : \phi \rightsquigarrow C \\ \overline{v}' = \mathrm{fuv}(\phi, C) - \mathrm{fuv}(\Gamma)\end{array} \qquad \begin{array}{l}\Gamma \vdash e_i : \sigma_i \rightsquigarrow C_i \\ \overline{v_i} = \mathrm{fuv}(\sigma_i, C_i) - \mathrm{fuv}(\Gamma, \phi, C)\end{array} \qquad \overline{\alpha}, \overline{\beta} \text{ fresh}}{\begin{array}{l}\Gamma \vdash (h\, e_1 \ldots e_n :: \forall \overline{b}.\, \eta) : \forall \overline{b}.\, \eta \\ \rightsquigarrow\ \forall \overline{b}.\, \exists \overline{\alpha^{\mathfrak{u}}}\, \overline{\beta^{\mathfrak{u}}}\, \overline{v}'.\left(\begin{array}{llll} C & \wedge\ \ \phi \leqslant^{n}_{\mathfrak{u}} \alpha^{\mathfrak{u}}_1 \to \beta^{\mathfrak{u}}_1 & \wedge\ \ \forall\{v_1\}.\, C_1 \Rightarrow \sigma_1 \leq \alpha^{\mathfrak{u}}_1 & \\ & \wedge\ \ \beta^{\mathfrak{u}}_1 \leqslant^{n-1}_{\mathfrak{u}} \alpha^{\mathfrak{u}}_2 \to \beta^{\mathfrak{u}}_2 & \wedge\ \ \forall\{v_2\}.\, C_2 \Rightarrow \sigma_2 \leq \alpha^{\mathfrak{u}}_2 & \\ & \wedge\ \ \vdots & \wedge\ \ \vdots & \wedge\ \ \beta^{\mathfrak{u}}_n \leqslant^{0}_{\mathfrak{u}} \eta \end{array}\right)\end{array}}\ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_1 : \phi \rightsquigarrow C_1 \qquad \Gamma, x : \phi \vdash e_2 : \sigma \rightsquigarrow C_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma \rightsquigarrow C_1 \wedge C_2}\ \textsc{Let}$$

Figure 8. Constraint generation

$$\boxed{\mathrm{freshen}^{n}_{\mathfrak{s}}(\sigma) \implies \langle \overline{v}, \mu \rangle}$$

$$\frac{\mu \rhd^{n}_{\mathfrak{s}} \Delta \qquad \overline{\alpha} \text{ fresh} \qquad v_i = \alpha_i^{\Delta(a_i)}}{\mathrm{freshen}^{n}_{\mathfrak{s}}(\forall \overline{a}.\, \mu) \implies \langle \overline{v}, [\overline{a \mapsto v}]\mu \rangle}$$

$$\boxed{C\ ;\ \overline{v} \implies C'\ ;\ \overline{v}'}$$

$$[\textsc{conj}]\ \frac{C_1\ ;\ \overline{v} \implies C_1'\ ;\ \overline{v}'}{C_1 \wedge C_2\ ;\ \overline{v} \implies C_1' \wedge C_2\ ;\ \overline{v}'} \qquad [\textsc{forall}]\ \frac{C\ ;\ \overline{v}_{\mathrm{in}} \implies C'\ ;\ \overline{v}'_{\mathrm{in}}}{\forall \overline{a}.\, \exists \overline{v}_{\mathrm{in}}.\, C\ ;\ \overline{v} \implies \forall \overline{a}.\, \exists \overline{v}'_{\mathrm{in}}.\, C'\ ;\ \overline{v}}$$

[EQREFL]	$\sigma \sim \sigma\ ;\ \overline{v}$	$\implies$	$\top\ ;\ \overline{v}$	
[EQMONO]	$\mathsf{T}\,\sigma_1 \ldots \sigma_n \sim \mathsf{T}\,\phi_1 \ldots \phi_n\ ;\ \overline{v}$	$\implies$	$\sigma_1 \sim \phi_1, \ldots, \sigma_n \sim \phi_n\ ;\ \overline{v}$	
[EQSUBST]	$(\alpha^{\mathfrak{s}} \sim \sigma) \wedge C\ ;\ \overline{v}$	$\implies$	$(\alpha^{\mathfrak{s}} \sim \sigma) \wedge [\alpha^{\mathfrak{s}} \mapsto \sigma]C\ ;\ \overline{v}$	if $\vdash \sigma : \mathfrak{s}$, and $\alpha \notin \mathrm{ftv}(\sigma)$
[EQVAR]	$\alpha^{\mathfrak{s}_1} \sim \beta^{\mathfrak{s}_2}\ ;\ \overline{v}$	$\implies$	$\beta^{\mathfrak{s}_2} \sim \alpha^{\mathfrak{s}_1}\ ;\ \overline{v}$	if $\mathfrak{s}_1 \sqsubset \mathfrak{s}_2$
[EQFULLY]	$\alpha^{\mathfrak{m}} \sim \sigma\ ;\ \overline{v}$	$\implies$	$\{\beta^{\mathfrak{s}} \sim \gamma^{\mathfrak{m}} \mid \beta^{\mathfrak{s}} \in \mathrm{fuv}(\sigma), \mathfrak{s} \neq \mathfrak{m}\}\ ;\ \overline{v}, \overline{\gamma^{\mathfrak{m}}}$	
[INSTMONO]	$\mu \leqslant^{n}_{\mathfrak{s}} \eta\ ;\ \overline{v}$	$\implies$	$\mu \sim \eta\ ;\ \overline{v}$	
[INST∀L]	$(\forall \overline{a}.\, \mu) \leqslant^{n}_{\mathfrak{s}} \eta\ ;\ \overline{v}$	$\implies$	$\mu' \sim \eta\ ;\ \overline{v}, \overline{v}'$	where $\mathrm{freshen}^{n}_{\mathfrak{s}}(\forall \overline{a}.\, \mu) \implies \langle \overline{v}', \mu' \rangle$
[INST∀L]	$(\forall\{\overline{v}'\}.\, \overline{C} \Rightarrow \sigma) \leq \eta\ ;\ \overline{v}$	$\implies$	$\overline{C} \wedge \left(\sigma \leqslant^{0}_{\mathfrak{m}} \eta\right)\ ;\ \overline{v}, \overline{v}'$	
[INST∀R]	$\mathfrak{g} \leq (\forall \overline{a}.\, \mu)\ ;\ \overline{v}$	$\implies$	$\forall \overline{a}.\, (\mathfrak{g} \leq \mu)\ ;\ \overline{v}$	

Figure 9. Solving rules

this is not problematic, since on application type variables are instantiated and regeneralized using the $\leq$ relation.

4.3.2 Instantiation and generalisation constraints

For instantiation constraints $\leqslant^n_{\mathfrak{s}}$, Rule INSTMONO encodes the fact that in our system if two top-level monomorphic types μ, η are in an instance relation, they must be equal. This is a consequence of the invariance of type constructors. INST∀L instantiates a polytype σ with fresh unification variables, much as in the usual Damas-Milner algorithm, except that we must use a sort-respecting instantiation. This is done by $\text{freshen}^n_{\mathfrak{s}}$, which in turn uses the already-introduced classification judgment $\triangleright^n_{\mathfrak{s}}$ (Figure 4). Finally, the new variables enter the set of existentially quantified variables. Notice that the right hand side of an instantiation constraint $\sigma \leqslant^n_{\mathfrak{s}} \mu$ is always a top-level monomorphic type μ, so we do not need to worry about having a polymorphic type on the right.

Finally, we come to generalisation constraints, which (recall Section 4.1) express a deferred generalisation decision. Rule INST∀L is simple: if the right hand side has no top-level foralls (it is of the form η) then there is no generalisation to be done, so it suffices to release all the captured constraints C and existentials υ' into the current constraint.

INST∀R is where actual generalisation takes place. In order to forge some intuition, let us look at the constraint

$$(\forall\{\alpha\,\beta\}.\,\alpha \leqslant^0_{\mathfrak{m}} \beta \Rightarrow \alpha \to \beta) \leq (\forall p.\, p \to p)$$

This generalisation constraint says that by performing some solving and possibly abstracting over some of the variables α and β, we should get the polymorphic type $\forall p.\, p \to p$. Following standard practice, we skolemise the type on the right, introducing a fresh skolem or rigid variable p, which should not be unified.

$$\alpha \leqslant^0_{\mathfrak{m}} \beta \quad \wedge \quad \alpha \to \beta \sim p \to p$$

We obtain a solution by making $\alpha \sim \beta \sim a$. But in order for this solution to remain valid, we must guarantee that the skolem p *does not escape* to the outer world. We recall this restriction by means of a fresh quantification constraint $\forall p.\exists \alpha\,\beta.(\alpha \leqslant^0_{\mathfrak{m}} \beta \quad \wedge \quad \alpha \to \beta \sim a \to a)$. Rule INST∀R achieves this rather neatly simply by doing skolemisation and pushing the $\mathfrak{g}$ inside; then INST∀L will do the rest.

The rules applicable to instantiation and generalisation constraints do not handle every case. In particular, whenever an unrestricted variable appears in one of the sides of the constraint, there are good reasons to wait:

1. If we have $\alpha^{\mathrm{u}} \leqslant^n_{\mathfrak{s}} \mu$ we cannot turn it directly into $\alpha^{\mathrm{u}} \sim \mu$, because α^{u} might be unified later to a polymorphic type and we need instantiation.
2. Similarly, if we have $(\forall\{\overline{\alpha}\}.\, C \Rightarrow \mu) \leq \alpha^{\mathrm{u}}$, and α^{u} is later substituted by a polytype, we must skolemise.

The guardedness restrictions are carefully crafted to ensure that the solver is never completely *stuck*, unless the constraint set as a whole is inconsistent. A single constraint can be stuck for some time, but if we are dealing with a consistent constraint set, the promise is that it will, by solving steps applied to other constraints in the set, become unstuck.

Theorem 4.1. *Suppose $\Gamma \vdash e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a new set C' without instantiation and generalisation constraints which fixes the value of all unrestricted and top-level monomorphic variables.*

The resulting constraint set is an instance of the problem of first-order unification under a mixed prefix. [3] (which in our system is expressed by quantification constraints). A complete algorithm to solve this kind of problem is described by Pottier and Rémy [15]. As a consequence, we have:

Corollary 4.2. *Suppose $\Gamma \vdash e : \sigma \rightsquigarrow C$. Then C is either inconsistent, or can be rewritten to a solved form.*

Unfortunately, once we extend the language of types, by introducing type classes and local assumptions, completeness is known not to hold [21]. Furthermore, the approach by Pottier and Rémy [15] is no longer applicable. With that in mind, we introduce a *different* approach to solve the problem of unification under a mixed prefix, which *does* scale to handle local assumptions. The approach is rather simple – a single rule FLOAT in Figure 10 – and is directly inspired by how GHC handles constraints: by floating constraints F from inside a quantification constraint to outside. When can we do that? Precisely when the constraint does not mention the skolems. But what about the existentials? For example, suppose we have

$$\exists \alpha. \ldots (\forall a.\exists \beta.(\alpha \sim [\beta]) \wedge C) \ldots$$

We would like to float the constraint $(\alpha \sim [\beta])$ out of the quantification constraint, but then β would be out of scope. We can solve this by "promoting" β: producing a fresh β' that lives in the outer scope, and making β equal to it, thus:

$$\exists \alpha, \beta'. \ldots (\alpha \sim [\beta'])) \wedge (\forall a.\exists \beta.(\beta \sim \beta' \wedge C)) \ldots$$

All this is expressed directly by rule FLOAT. If we cannot float, we have a skolem escape error; for example, consider:

$$\exists \alpha. \ldots (\forall a.\exists \beta.(\alpha \sim [a]) \wedge C) \ldots$$

Here we cannot float $(\alpha \sim [a])$ because it mentions the skolem a, so an inner skolem has leaked into an outer scope (α is bound further out). Floating makes manifest that skolem escape has not happened, and brings the constraint nearer to *solved form*, which we treat next.

Conjecture 4.3. *The solver presented in Figure 9 is complete for unification problems under a mixed prefix.*

We stress that for constraints without local assumptions we can either choose the algorithm in Pottier and Rémy [15] – which is known to be complete and thus allows us to prove

$$[\textsc{float}]\ \frac{\mathrm{ftv}(F) \cap \overline{a} = \emptyset \qquad \overline{\alpha^{\mathsf{s}}} = \mathrm{fuv}(F) \cap \upsilon_{\mathsf{in}} \qquad \overline{\gamma^{\mathsf{s}}}\ \text{fresh} \qquad \overline{E} = \bigwedge_{\alpha^{\mathsf{s}} \in \overline{\alpha}} \alpha^{\mathsf{s}} \sim \gamma^{\mathsf{s}}}{\forall \overline{a}.\, \exists \overline{\upsilon}_{\mathsf{in}}.\, (C \wedge F) \,;\, \overline{\upsilon} \implies [\overline{\alpha^{\mathsf{s}} \mapsto \gamma^{\mathsf{s}}}]F \wedge \forall \overline{a}.\, \exists \overline{\upsilon}_{\mathsf{in}}.\, (C \wedge E) \,;\, \overline{\upsilon}, \overline{\gamma^{\mathsf{s}}}}$$

Figure 10. Solving rule for quantification constraints

$$\frac{\vdash \sigma : \mathfrak{s} \qquad \mathrm{ftv}(\sigma) \subseteq \overline{a} \cup \overline{\alpha}}{\overline{a}\,;\,\overline{\alpha}\,;\,\{\beta\} \vdash \beta^{\mathsf{s}} \sim \sigma\ \text{solved}}\ \textsc{SolvedVar}$$

$$\frac{\overline{a}\,;\,\overline{\alpha}\,;\,\overline{\beta}_1 \vdash C_1\ \text{solved} \qquad \overline{a}\,;\,\overline{\alpha}\,;\,\overline{\beta}_2 \vdash C_2\ \text{solved}}{\overline{a}\,;\,\overline{\alpha}\,;\,\overline{\beta}_1 \uplus \overline{\beta}_2 \vdash C_1 \wedge C_2\ \text{solved}}\ \textsc{SolvedConj}$$

$$\frac{\overline{\upsilon} = \overline{\gamma}_1 \uplus \overline{\gamma}_2 \qquad \overline{a} \cup \overline{b}\,;\,\overline{\alpha} \cup \overline{\gamma}_1\,;\,\overline{\gamma}_2 \vdash C\ \text{solved}}{\overline{a}\,;\,\overline{\alpha}\,;\,\emptyset \vdash \forall \overline{b}.\, \exists \overline{\upsilon}.C\ \text{solved}}\ \textsc{SolvedQuant}$$

Figure 11. Definition of solved set of constraints

a completeness theorem (Theorem 4.6) – or the one with the FLOAT rule – which we only conjecture as complete but scales when the constraint language is extended.

4.3.3 Solved form

A constraint is in *solved form* if it consists only of quantification and equality constraints ($\upsilon \sim \sigma$); and the equalities constitute a well-sorted idempotent substitution of its unification variables. For example

$$\exists \alpha^{\mathsf{m}}.(\alpha^{\mathsf{m}} \sim \mathsf{Int}) \wedge (\forall b.\exists \beta^{\mathsf{u}}.\, \beta^{\mathsf{u}} \sim (b \to \mathsf{Int}))$$

is in solved form. Being in solved form is more than just syntactic; here are two constraints that are not:

$$\exists \alpha.(\alpha \sim \mathsf{Int}) \wedge (\alpha \sim \mathsf{Bool}) \qquad \exists \alpha.\ldots(\forall b.\alpha \sim [b])\ldots$$

In the first there are two equalities for α (we should apply EQSUBST to make progress); in the second, there is a skolem-escape problem. However it is OK for a unification variable to have *no* equalities; it is simply unconstrained.

Figure 11 defines solved form precisely. We keep a set of variables $\overline{\beta}$ for which we ensure that there is precisely one equality constraint, and another set $\overline{\alpha}$ (the unconstrained variables) for which there are none. Rule SolvedVar expects precisely one β, checks well-sortedness, and also checks that σ does not mention any variables other than the skolems and unconstrained unification variables – the latter check ensures idempotence.

Rule SolvedConj partitions the $\overline{\beta}$ between the two conjunctions. Rule SolvedQuant partitions the local existentials $\overline{\upsilon}$ into the unconstrained sets, γ_1 and γ_2 resp.

4.4 Soundness, principality and completeness

The inference algorithm presented here satisfies the usual properties of soundness, principality, and completeness with respect to the declarative specification. In order to state these results, we need the auxiliary notion of substitution induced by a solved form.

$$\frac{C_{\mathsf{s}} = E \wedge R, \text{ where } E \text{ are all the equalities in } C_{\mathsf{s}}}{\widehat{C_{\mathsf{s}}} = [\overline{\alpha \mapsto \sigma} \mid \alpha \sim \sigma \in E]}$$

The proofs are given in Appendix E.2.

Theorem 4.4 (Soundness)**.** *Let Γ be a closed environment and e an expression. If $\Gamma \vdash e : \sigma \rightsquigarrow C$ and C_{s} is a solution for C with an induced substitution $\widehat{C_{\mathsf{s}}}$, then we have $\Gamma \vdash e : \widehat{C_{\mathsf{s}}}(\sigma)$.*

Theorem 4.5 (Principality)**.** *Suppose $\Gamma \vdash e : \sigma$. Then there exists a type $\sigma^\star$ such that $\Gamma \vdash e : \sigma^\star$ and $\sigma = \pi\sigma^\star$ where π is a fully monomorphic substitution.*

Theorem 4.6 (Completeness)**.** *Let Γ be a closed environment and e an expression. If $\Gamma \vdash e : \sigma$ then $\Gamma \vdash e : \phi \rightsquigarrow C$ and C can reach a solved form.*

5 Practical matters

Prototype. We have implemented a prototype of the type inference process described in this paper, including support for Haskell's type classes by extending GI as described in Appendix B. The expressions in Figure 2 are accepted or rejected as described by the table, in the GI decl. column.

Backward compatibility. GI does not support any co- or contra-variance in function types, nor deep skolemization; but GHC does, at least with the *RankNTypes* extension. One might worry that many existing Haskell libraries would need to be modified. To quantify this impact, we modified GHC to impose those restrictions and rebuilt all the packages in Stackage which require the *RankNTypes* extension. In order to minimize the annotation burden, we added a simple special case that supports function definition with a forall to the right of an arrow:

$f :: \forall\, a.\ a \to (\forall\, b.\ b \to b)$
$f\ x\ y = y$

Such definitions are very common in libraries such as Scrap Your Boilerplate. With this done, very few packages required modifications, and modifications were always η-expansions. Consider (*flip* f), where *flip*'s type is in Figure 1. This is ill-typed because *flip* requires an argument of type $a \to b \to c$, but f's type, after instantiation, looks like $\tau \to \forall\, b.\ b \to b$. The fix is simple: just η-expand the argument, thus (*flip* $(\lambda x \to f\ x)$). GHC does this automatically at the moment, but in fact this η-expansion is unsound in general, because of *seq*, so requiring manual η-expansion is probably the right choice anyway.

Of the 2,400 packages in Stackage, 609 use *RankNTypes*; of these, only 75 required manual changes, all of which were simple η-expansions. One (*singletons*) would require larger changes, becuase it uses Template Haskell to *generate* Haskell code; so it needs to generate η-expanded code. Two more failed for reasons we have yet to investigate. Our conclusion is that the impact of our proposed changes is extremely minor, especially since GHC's current covert η-expansion strategy is unsound in the first place.

Relaxed inference. Section 2.2 describes some limitations of our system, like the impossibility of inferring a polymorphic element type for the empty list [ ]. There are some scenarios, though, where one can relax the guardedness restrictions and still obtain a correct solution for the generated constraints. We describe that *relaxed* solver in Appendix C. There is a net gain on the amount of programs accepted by this approach, as Figure 2 shows under the GI relax. heading.

6 Related work

Full type inference for System F is undecidable [24] – partial type inference with known generalisation positions but unknown instantiations can be reduced to h.o. unification [14]. System F lacks principal types, making modular type inference and the addition of ML-style let-bindings impossible. Higher-rank type inference with instantiation restricted to *monomorphic types* has some successful solutions [4, 13, 17], exploiting a mix of annotation propagation and unification.

On the other hand, no solution for impredicativity with a good benefit-to-weight ratio has been presented to-date. MLF [1, 2, 18] is an *extension* of System F based on quantification with instance bounds. The resulting system is powerful, but also quite complex to implement; in return we get back principal types. There have been several attempts to simplify the user-facing part of MLF to System F types. FPH [23] exposes a "box" structure around inferred types (that would be hidden under a constraint in MLF). Flexible types [9] avoid quantification over equality constraints. Implementing these systems in a working compiler is a significant undertaking, and so is the integration with features like type classes[10].

For this reason, there are proposals for algorithms simpler than MLF. Boxy Types [22] is an early attempt to push bidirectional type inference to allow impredicativity, but resulted in a complex specification. HMF [8] imposes universal conditions on typing derivations to recover principal types. QML [19], inspired by boxed polymorphism [12, 16], introduces an only-explicitly-instantiable $\forall$. Recent work [5] also proposes a distinction between an implicitly and an explicitly instantiable $\forall$; only the latter is impredicative.

We return now to Figure 2, where we present a collection of examples appearing in selected related works on type inference for impredicativity, namely MLF [1], HMF [8], FPH [23], HML [9]. We have selected those systems to compare because they strike extremely well in expressivity and require very few type annotations. The table shows the flexibility/expressivity price we pay in order to keep the implementation and specification costs low, and avoid the introduction of new type system features (such as types with constraints or boxes) or new forms of annotations (such as QML-style instantiation annotations). We also show the annotation that can recover the typeability of a program in cases where a valid type exists. As one observes, MLF can type all of programs that do not require implicit η-expansion ($k\ \ h\ \ lst$) or the use of a polymorphic function argument at two types $\lambda f.\,(f\ 1, f\ \mathit{True})$. Unsurprisingly, HML is only minimally less powerful as it cannot type $\lambda xs.\ poly\ (head\ xs)$ because *xs* would have to be assigned a polymorphic type, even though it's only used at this type. FPH is equally expressive for the applicative fragment of System F – however its treatment of λ-abstractions requires the returned type to be a fully-resolved top-level monomorphic type and hence fails to type check *choose id auto* because *auto* will be assigned the same type as GI infers ($(\forall\, a.\, a \to a) \to b \to b$). HMF on the other hand is just based on local decisions about polymorphic instantiations and – without an extension to n-ary applications – fails to type check programs where the local instantiation has to be delayed to take more arguments into account (e.g. fails to type check *id* : *ids*). The relaxed GI system on the other hand can – modulo the quirk about not generalizing the bodies of lambda abstractions that only MLF and HML can tackle – type check the same programs as those systems. GI (vanilla) is more restrictive than those systems but incomparable to HMF. To determine why a program fails to type check (and how it should be fixed) it suffices to determine whether some function has been instantiated to a type with top-level polymorphism in its arguments. For example, $f\ (choose\ id)\ ids$ fails to type check because it requires the instantiation of $choose :: \forall\, a.\, a \to a \to a$ to $(\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a)$ and none of the arguments have a type with a top-level constructor. The fix is to add an annotation around the offending expression to determine its type: $f\ \ (choose\ \ id :: (\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a))\ \ ids$. The relaxed GI recovers pretty much of the lost expressivity, and is by construction as expressive as the vanilla GI, but we have yet to determine a simple declarative specification.

7 Further work

GI is relatively simple and predictable but, as mentioned earlier, we would like it to be just a bit more expressive. For example, the need to type-annotate occurrences of the empty list (Section 2.2) seems particularly tiresome – and it is hard to see why it should truly be necessary. Thus motivated, we are investigating some modest extensions, both of the declarative system and of the constraint solver, that would accept more programs. The swamp is calling!

References

[1] Didier Le Botlan and Didier Rémy. 2003. ML^F: raising ML to the power of system F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, Colin Runciman and Olin Shivers (Eds.). ACM, 27–38. https://doi.org/10.1145/944705.944709

[2] Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Inf. Comput.* 207, 6 (2009), 726–785.

[3] Hubert Comon and Pierre Lescanne. 1988. *Equational problems and disunification.* Research Report RR-0904. INRIA. https://hal.inria.fr/inria-00075652

[4] Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. https://doi.org/10.1145/2500365.2500582

[5] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.* Springer-Verlag New York, Inc., New York, NY, USA, 229–254. https://doi.org/10.1007/978-3-662-49498-1_10

[6] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Scripting the type inference process. *SIGPLAN Notices* 38, 9 (2003), 3–13. https://doi.org/10.1145/944746.944707

[7] James Hook and Peter Thiemann (Eds.). 2008. *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008.* ACM.

[8] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism, See [7], 283–294. https://doi.org/10.1145/1411204.1411245

[9] Daan Leijen. 2009. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 66–77. https://doi.org/10.1145/1480881.1480891

[10] Daan Leijen and Andres Löh. 2005. Qualified types for MLF. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 144–155. https://doi.org/10.1145/1086365.1086385

[11] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282. https://doi.org/10.1145/357162.357169

[12] J. W. O'Toole, Jr. and D. K. Gifford. 1989. Type Reconstruction with First-class Polymorphic Values. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89).* ACM, New York, NY, USA, 207–217. https://doi.org/10.1145/73141.74836

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

[14] Frank Pfenning. 1988. Partial Polymorphic Type Inference and Higher-order Unification. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88).* ACM, New York, NY, USA, 153–163. https://doi.org/10.1145/62678.62697

[15] François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

[16] Didier Rémy. 1994. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '94).* Springer-Verlag, London, UK, UK, 321–346. http://dl.acm.org/citation.cfm?id=645868.668492

[17] Didier Rémy. 2005. Simple, Partial Type-inference for System F Based on Type-containment. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05).* ACM, New York, NY, USA, 130–143. https://doi.org/10.1145/1086365.1086383

[18] Didier Rémy and Boris Yakobowski. 2008. From ML to ML^F: graphic type constraints with efficient type inference, See [7], 63–74. https://doi.org/10.1145/1411204.1411216

[19] Claudio V. Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-class Polymorphism for ML. In *Proceedings of the 2009 ACM SIGPLAN Workshop on ML (ML '09).* ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1596627.1596630

[20] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[21] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X): Modular type inference with local assumptions. *J. Funct. Program.* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

[22] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 251–262. https://doi.org/10.1145/1159803.1159838

[23] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell, See [7], 295–306. https://doi.org/10.1145/1411204.1411246

[24] J. B. Wells. 1993. *Typability and Type Checking in the Second-Order Lambda-Calculus Are Equivalent and Undecidable.* Technical Report. Boston, MA, USA.

[25] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2015. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 12–21. https://doi.org/10.1145/2737924.2738009