Guarded impredicative polymorphism

ANONYMOUS AUTHOR(S)

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.

ACM Reference format:
Anonymous Author(s). 2017. Guarded impredicative polymorphism. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 39 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Type inference for impredicative polymorphism is a deep, deep swamp. The literature is crowded with corpses (including several of our own): type systems for impredicative polymorphism that never quite worked well enough to be deployed in a production compiler [Botlan and Rémy 2003; Leijen 2008, 2009; Vytiniotis et al. 2006, 2008]. 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 2.2-2.4), 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 [Botlan and Rémy 2003; Leijen 2009; Vytiniotis et al. 2008]), nor do we introduce new forms of annotations [Russo and Vytiniotis 2009] or side-conditions that require principal types [Leijen 2008].

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{array}{ll}
\mathit{head} & :: \forall p.\, [p] \to p \\
\mathit{tail} & :: \forall p.\, [p] \to [p] \\
[\,] & :: \forall p.\, [p] \\
(:) & :: \forall p.\, p \to [p] \to [p] \\
\mathit{single} & :: \forall p.\, p \to [p] \\
(+\!\!+) & :: \forall p.\, [p] \to [p] \to [p] \\
\mathit{length} & :: \forall p.\, [p] \to \mathit{Int} \\
\mathit{map} & :: \forall p\ q.\, (p \to q) \to [p] \to [q] \\
\mathit{ids} & :: [\forall a.\, a \to a]
\end{array}
\qquad
\begin{array}{ll}
\mathit{id} & :: \forall a.\, a \to a \\
\mathit{inc} & :: \mathit{Int} \to \mathit{Int} \\
\mathit{twice} & :: \forall a.\, (a \to a) \to a \to a \\
\mathit{choose} & :: \forall a.\, a \to a \to a \\
\mathit{poly} & :: (\forall a.\, a \to a) \to (\mathit{Int}, \mathit{Bool}) \\
\mathit{auto} & :: (\forall a.\, a \to a) \to (\forall a.\, a \to a) \\
\mathit{app} & :: \forall a\ b.\, (a \to b) \to a \to b \\
\mathit{revapp} & :: \forall a\ b.\, a \to (a \to b) \to b \\
\mathit{runST} & :: (\forall s.\, \mathit{ST}\ s\ v) \to v \\
\mathit{argST} & :: \forall s.\, \mathit{ST}\ s\ \mathit{Int}
\end{array}
$$

Fig. 1. Type signatures for functions used in the text

- 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, **do** notation, and pattern matching (Section 3.4). 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 3.6). The resulting system can express any System F program, as we show in Figure 8.
- 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, which is already based in constraints. Type-correct programs can easily be elaborated into System FC, GHC's intermediate language, without extensions. Our inference algorithm scales readily to handle GADTs and type classes, as we show in Section 5. Higher kinds, type-level functions and other type system features pose no additional problem. Moreover, as we discuss in Section 6, 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 7. 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.

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 $\mathit{single} :: \forall p.\, p \to [p]$. 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? This is clearly a *hard* case for type inference because it requires information from the context. In GI our approach is to exploit the easy case and turn the hard case to easy by requiring type annotations; finally when annotations are not present we will fall back to monomorphic instantiation. We hope thereby to get 90% of the joy for 10% of the pain. Specifically:

- 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.
- For the choices of impredicative instantiation, we accept a particular information flow in an n-ary call: start with a function with a known type; guide its instantiation by the types of its explicitly-supplied arguments; and do not attempt to use the context of the call to guide instantiation.
- We instantiate a function with a polymorphic type (i.e. impredicatively) only if the polymorphism is "guarded", either in the type of the function (as we saw with (*head xs*) above), or the type of the argument.

What does this last bullet mean? Consider the call (:) *id ids* (see Figure 1 for type signatures). Here the variable p in (:)'s type appears under a list type constructor in the second argument, so its instantiation is fixed by the second argument *ids*. All is good.

But what if we had a partially-applied call (:) *id*? Now p does not appear guarded (under a type constructor) in the first argument of (:)'s type. And indeed, it is not immediately clear whether to instantiate p with $\forall\, a.\, a \to a$, or $Int \to Int$, etc; it is like the (*single id*) case. We do not reject this program, of course; rather we just insist that p is instantiated with a monotype. So for us, the notion of "guardedness" clearly involves the number of arguments supplied in an n-ary call.

Similarly, consider the call *map poly ids*. Here, p (in *map*'s type) appears under a type constructor in the second argument of *map*, so this application is fine despite the polymorphism in *poly*'s type. What about the partially-applied call (*map poly*)? In GI that too is OK, because we treat $(\to)$ as invariant, like any other type constructor. This occasionally means that you need to do some eta-expansion (Section 3.2), but it makes more type variables guarded, and that is very worthwhile. But a call of *map* applied to no arguments (for example, it might be passed as an argument to another function) cannot be instantiated impredicatively.

Up to now we have spoken of guardedness as a property of the function's type, given the number of arguments to which it is applied. However, in GI guardedness is not *only* a property of the type of the function; it *also* involves the types of its arguments. Consider the call *single xs*, where $xs :: [\forall\, a.\, a \to a]$. Now there is no doubt at what type to instantiate $single :: \forall\, p.\, p \to [p]$! It must be instantiated with $p := [\forall\, a.\, a \to a]$. Why is there no doubt? Again because the scary polymorphic type appears under a list constructor.

2.2 A general rule for n-ary applications

Here is the general rule:

> Rule 1: *When instantiating a type at an n-ary call, a type variable may only be assigned a polymorphic type σ if, in the instantiated type, σ appears under a type constructor in one or more of the supplied arguments.*

This rule is carefully crafted. To illustrate, consider the examples (again, 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 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 to match. But Rule 1 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.

Rule 1 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.

2.3 Ignoring the context of a call

Notice that Rule 1 *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, Rule 1 allows the programmer to understand impredicativity in a simple bottom-up way. For example, consider the well-typed expression (*map head* (*single ids*)) (Figure 2)? In our type system, the types for *head* and *single ids* are determined, in fact, 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 6, and we also introduce annotations to cover the remaining cases.

2.4 Lambdas

In common with many other approaches to impredicativity, we take a conservative position on lambda-bound variables. Consider $g\ (\lambda f.\ (f\ \texttt{'x'}, f\ True))$, 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:

> Rule 2: *Every lambda abstraction whose argument is polymorphic must be annotated. If is it 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 6 we explore a way 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 Rule 2 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 $(\forall\ a.\ a \to a).\ x\ x$, 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 $(\forall a.\, a \to a).\, (x\ x :: \forall a.\, a \to a)$ instead. (Note that this example also shows up in Figure 2 as *auto*.)

2.5 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 7.

One 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 $(runST\ \$\ \mathbf{do}\ \{\ldots\})$ (the type of *runST* is given in Figure 1). 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 sometimes forces programmers to define new types as a work around. Consider the following from Haskell's `lens` library[1]:

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

Because *Lens* is a polymorphic type, forming a list of lenses $[Lens\ s\ t\ a\ b]$ is illegal in Haskell, because you cannot instantiate the list constructor (:) at a polymorphic type. So programmers have to resort to additional wrapping and unwrapping

$$\mathbf{newtype}\ ReifiedLens\ s\ t\ a\ b = Lens\ \{runLens :: Lens\ s\ t\ a\ b\}$$

Now you can build a list of reified lenses, of type $[ReifiedLens\ s\ t\ a\ b]$, but doing so is tiresome, because it requires many explicit conversions between *Lens* and *ReifiedLens*. It would be much better if *Lens* can be treated as a first-class type, and that is exactly what impredicativity allows.

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 base to understand when and why annotations are needed in their programs.

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 0-ary application. Because of Rule 2 (Section 2.4), 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.

[1] https://hackage.haskell.org/package/lens-4.15.3/docs/Control-Lens-Reified.html

	GI decl. spec.	GI relaxed	MLF	HMF	FPH	HML
POLYMORPHIC INSTANTIATION: given $f :: \forall\, a.\, (a \to a) \to [a] \to a$						
$const2 = \lambda x\ y.\ y$	✓	✓	✓	✓	✓	✓
MLF infers $(b \geqslant \forall\, c.\, c \to c) \Rightarrow a \to b$, whereas GI infers $a \to b \to b$.						
$choose\ id$	✓	✓	✓		✓	✓
MLF/HML infer $(a \geqslant \forall\, b.\, b \to b) \Rightarrow a \to a$, whereas FPH/GI infer $(a \to a) \to a \to a$.						
$choose\ [\,]\ ids$	No	✓	✓	✓	✓	✓
GI needs an annotation on $[\,] :: [\forall\, a.\, a \to a]$.						
$auto = \lambda(x :: \forall\, a.\, a \to a).\ x\ x$	✓	✓	✓	No	✓	✓
MLF infers $(\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a)$, whereas GI infers $(\forall\, a.\, a \to a) \to b \to b$.						
$id\ auto$	✓	✓	✓	✓	✓	✓
$choose\ id\ auto$	No	No	✓	No	No	✓
Relaxed GI accepts the version with $auto :: (\forall\, a.\, a \to a) \to (\forall\, a.\, a \to a)$.						
$f\ (choose\ id)\ ids$	No	✓	✓	No	✓	✓
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 OPERATING ON POLYMORPHIC LISTS: given $g :: \forall\, a.\, [a] \to [a] \to a$						
$length\ ids$	✓	✓	✓	✓	✓	✓
$tail\ ids$	✓	✓	✓	✓	✓	✓
$head\ ids$	✓	✓	✓	✓	✓	✓
$single\ id$	✓	✓	✓	✓	✓	✓
$id : ids$	✓	✓	✓	No	✓	✓
$(\lambda x.\, x) : ids$	✓	✓	✓	No	✓	✓
$single\ inc +\!\!+ single\ id$	✓	✓	✓	✓	✓	✓
$g\ (single\ id)\ ids$	No	No	✓	No	✓	✓
$map\ poly\ (single\ id)$	No	No	✓	✓	✓	✓
GI needs an annotation on $single\ id :: [\forall\, a.\, a \to a]$ in the two previous examples.						
$map\ head\ (single\ ids)$	✓	✓	✓	✓	✓	✓
APPLICATION FUNCTIONS						
$app\ poly\ id$	✓	✓	✓	✓	✓	✓
$revapp\ poly\ id$	✓	✓	✓	✓	✓	✓
$runST\ argST$	✓	✓	✓	✓	✓	✓
$app\ runST\ argST$	✓	✓	✓	✓	✓	✓
$revapp\ runST\ argST$	✓	✓	✓	✓	✓	✓
η-EXPANSION: given $h :: Int \to \forall\, a.\, a \to a$, $k :: \forall\, a.\, a \to [a] \to a$, $lst :: [\forall\, a.\, Int \to a \to a]$,						
$k\ h\ lst$	No	No	No	No	No	No
$k\ (\lambda x.\, h\ x)\ lst$	✓	✓	✓	No	✓	✓

Fig. 2. Comparison of type systems (Figure 1 gives the types of the functions mentioned)

Skolem / rigid variables	$\mathbb{V}$	$\ni$	$a, b, c, \ldots$	
Type constructors	$\mathbb{T}$	$\ni$	$\rightarrow, \mathsf{T}, \mathsf{S}, \ldots$	
Fully monomorphic types	τ	$::=$	$a \mid \mathsf{T}\ \overline{\tau}$	
Top-level monomorphic 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	$\mathrm{ftv}(\sigma)$	$=$	free type variables of σ	

Fig. 3. Syntax of the language

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

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

$$\boxed{\sigma \rhd^n_{\mathfrak{s}} \Delta}$$

$$\frac{}{\mu \rhd^0_{\mathfrak{s}} [\mathrm{ftv}(\mu) \mapsto \mathfrak{s}]} \qquad \frac{\mu \rhd^n_{\mathfrak{s}} \Delta}{\forall \overline{a}.\, \mu \rhd^n_{\mathfrak{s}} \Delta \backslash \overline{a}}$$

$$\frac{\sigma_2 \rhd^{n-1}_{\mathfrak{s}} \Delta}{\sigma_1 \rightarrow \sigma_2 \rhd^n_{\mathfrak{s}} \Delta \sqcup [a \mapsto \text{ if } \sigma_1 \vdash a \text{ guarded then } \mathfrak{u} \text{ else } \mathfrak{t} \mid a \in \mathrm{ftv}(\sigma_1)]}$$

$$\boxed{\vdash \sigma : \mathfrak{s}} \qquad \boxed{\theta \text{ respects } \Delta}$$

$$\frac{}{\vdash \sigma : \mathfrak{u}} \qquad \frac{}{\vdash \mu : \mathfrak{t}} \qquad \frac{}{\vdash \tau : \mathfrak{m}} \qquad \frac{\text{for all } a \in \mathrm{dom}(\theta), \vdash \theta(a) : \Delta(a)}{\theta \text{ respects } \Delta}$$

Fig. 4. Definitions related to guardedness

- 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 semantics is as you would expect.

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}} x : \sigma}\ \textsc{VarHead} \qquad \frac{e \text{ not variable or application} \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash^{\mathsf{h}} e : \sigma}\ \textsc{ExprHead}$$

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

$$\frac{\Gamma \vdash e : \sigma \qquad \overline{b} \text{ fresh} \qquad \sigma \leqslant^{0}_{\mathfrak{m}} [\overline{a \mapsto b}]\mu}{\Gamma \vdash^{\mathsf{arg}} e : \forall \overline{a}.\, \mu}\ \textsc{ArgGen}$$

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

$$\frac{}{\mu \leqslant^{0}_{\mathfrak{s}} \mu}\ \textsc{InstMono} \qquad \frac{n \geqslant 1 \qquad \phi_2 \leqslant^{n-1}_{\mathfrak{s}} \sigma_2, \ldots, \sigma_n, \mu}{\phi_1 \to \phi_2 \leqslant^{n}_{\mathfrak{s}} \phi_1, \sigma_2, \ldots, \sigma_n, \mu}\ \textsc{InstArrow}$$

$$\frac{\mu \rhd^{n}_{\mathfrak{s}} \Delta \qquad \theta \text{ respects } \Delta \qquad \theta\mu \leqslant^{n}_{\mathfrak{s}} \sigma_1, \ldots, \sigma_n, \eta}{\forall \overline{a}.\, \mu \leqslant^{n}_{\mathfrak{s}} \sigma_1, \ldots, \sigma_n, \eta}\ \textsc{InstPoly}$$

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

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

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

Fig. 5. Declarative type system

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:

- If a appears under a type constructor (i.e. guarded) in at least one of the first n arguments of σ, then a may be instantiated by an unrestricted type ϕ.
- If a appears in one of the first n arguments of σ, a may be instantiated by a top-level monomorphic type μ.
- If a appears in the result type of σ, after stripping off n arguments, then a may be instantiated by a type of sort $\mathfrak{s}$.

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.

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

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

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

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

Fig. 6. Declarative type system with annotations in applications

scope with type ϕ. As discussed in Section 2.4, 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.

After typing the head, we instantiate the type of the head by means of the $\leqslant^{n}_{\mathfrak{s}}$ 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 considered *invariant*, even functions. This means that neither $[\forall\, a.\, a \to a] \not\leqslant^{n}_{\mathfrak{s}} [Int \to Int]$ nor $Int \to (\forall\, a.\, a \to a) \not\leqslant^{n}_{\mathfrak{s}} Int \to Int \to Int$.
- For function types the judgment $\leqslant^{n}_{\mathfrak{s}}$ 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. Continuing 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 $\leqslant^{2}_{\mathfrak{s}}$ 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 $\Gamma \vdash^{\mathsf{arg}} \lambda x.\, x : \forall\, a.\, a \to a$, as needed.

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

Proof. By inspection of the rules for the syntax-directed version of Hindley-Milner. Notice that all instantiations done in Hindley-Milner are admissible in GI, since the fully monomorphic types form a subset of the top-level monomorphic types which again form a subset of the fully polymorphic types. So even if a type variable is sorted as $\mathfrak{t}$ or $\mathfrak{u}$, an instance of sort $\mathfrak{m}$ can be chosen. □

3.3 Annotations in applications

With the rules in Figure 5, a single variable (not applied to any arguments) is treated as a 0-ary application. So rule App cannot get any guardedness information from its arguments, and so 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 $([\,] +\!\!+ \mathit{ids})$. 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 by the system, 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 $f = e :: \sigma$ to guarantee that the declaration is well-typed.

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, since the annotation tells us exactly what the types are those variables should be instantiated with.

Of course, annotated applications serve for arbitrary applications, not just 0-ary 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. One good consequence of this choice is that we can embed System F, as described in Section 3.5.

3.4 let bindings, do notation and pattern matching

A realistic language, such as Haskell [Marlow et al. 2010], includes other syntactic forms apart from those inherited from the λ-calculus. In some cases, those forms are defined as translations to the core language; **let** bindings and **do** notation are some important examples. As we shall see in this section, adopting the standard translation for these language constructs to derive their typing rule is not desirable: we can do better.

Let us start by looking at let-bindings. Following the long-established tradition, we could translate **let** $x = e_1$ **in** e_2 as $(\lambda x.\, e_2)\; e_1$, leading to the following derivation.

$$\frac{\dfrac{\Gamma, x : \tau \vdash e_2 : \sigma}{\Gamma \vdash \lambda x.\, e_2 : \tau \to \sigma} \qquad \tau \to \sigma \leqslant^1_{\mathfrak{m}} \tau, \mu \qquad \dfrac{\Gamma \vdash e_1 : \phi \qquad \phi \leqslant^0_{\mathfrak{m}} \tau}{\Gamma \vdash^{\mathsf{arg}} e_1 : \tau}}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \mu}$$

But this translation imposes an important restriction on the type of x: it must 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 7 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.

$$
\begin{array}{lcll}
\text{Data constructor} & & \ni & K \\
\textbf{do}\text{-statements} & s & ::= & e \mid x \leftarrow e\,;\,\overline{s} \mid \textbf{let}\ x = e\,;\,\overline{s} \\
\text{Expressions / terms} & e & ::= & \ldots \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{do}\ \overline{s} \mid \textbf{case}\ e_0\ \textbf{of}\ \{\overline{K\,\overline{x} \to e}\}
\end{array}
$$

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

$$\frac{\Gamma \vdash e_1 : \phi \qquad \Gamma, x : \phi \vdash e_2 : \sigma}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \sigma}\ \textsc{Let} \qquad \frac{\Gamma \vdash^{\text{do}}_M \overline{s} : M\,\phi}{\Gamma \vdash \textbf{do}\ \overline{s} : M\,\phi}\ \textsc{Do}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \qquad \sigma_0 \leqslant^0_{\text{m}} \mathsf{T}\,\overline{\phi_0} \qquad \begin{array}{l}\text{for each branch } K_i\,\overline{x_i} \to e_i \text{ with} \\ \underline{K_i : \forall \overline{a}\,\overline{b}.\,\overline{\sigma_i} \to \mathsf{T}\,\overline{a} \in \Gamma} \\ \Gamma, \overline{x_i : [\overline{a \mapsto \phi_0}]\sigma_i} \vdash e_i : \phi_\star\end{array}}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ \{\overline{K\,\overline{x} \to e}\} : \phi_\star}\ \textsc{Case}$$

$$\boxed{\Gamma \vdash^{\text{do}}_M e : \mu}$$

$$\frac{\Gamma \vdash^{\text{arg}} e : \mu}{\Gamma \vdash^{\text{do}}_M e : \mu} \qquad \frac{\Gamma \vdash^{\text{arg}} e : M\,\phi \qquad \Gamma, x : \phi \vdash^{\text{do}}_M \overline{s} : \eta}{\Gamma \vdash^{\text{do}}_M x \leftarrow e\,;\,\overline{s} : \eta} \qquad \frac{\Gamma \vdash e : \phi \qquad \Gamma, x : \phi \vdash^{\text{do}}_M \overline{s} : \eta}{\Gamma \vdash^{\text{do}}_M \textbf{let}\ x = e\,;\,\overline{s} : \eta}$$

Fig. 7. Declarative type system with **let**, **do** and pattern matching

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. [2011] 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** $x = (e_1 :: \lambda\phi)$ **in** e_2.

Translating **do** notation to a sequence of applications of $\gg=$ raises the same problems as **let**. The Haskell 2010 Language Report [Marlow et al. 2010] defines **do** $\{\,x \leftarrow e_1; e_2\,\}$ to be translated to $e_1 \gg= \lambda x \to e_2$. As before, this means that a fully monomorphic type has to be inferred for x, neglecting all information obtained from e_1. This restriction has been reported to forbid some classes of domain-specific languages [Augustsson 2011]. We give less restrictive rules in Figure 7, that again allow the type of x to be fully polymorphic.

Introducing pattern matching in the language gives no surprises, as witnessed by rule Case. We first need to check that the scrutinized expression e_0 can be given a type compatible with the indicated data constructor. It is important to note that the mere presence of the data constructors is enough to know the type constructor T, there is no inference at that point. Then we introduce new term variables in each branch, whose type is obtained by combining the type of the constructor with the inferred values for the type variables in T, namely $\overline{\phi_0}$. The return type of every branch should be equal, $\phi_\star$ (note that we do *not* allow the types of the branches to be instantiated).

3.5 Embedding of System F into GI

Our system is as expressive as System F, the gold standard for a fully-expressive impredicative type system:

Theorem 3.2 (Embedding of System F). *Let e be a System F expression. If $\Gamma \vdash^{\mathsf{F}} e_{\mathsf{F}} : \sigma \rightsquigarrow e'$, as defined in Figure 8, then $\Gamma \vdash e' : \sigma$ in* GI.

$$\text{System F terms} \quad e_F ::= x \mid \lambda(x :: \sigma).\, e_F \mid \Lambda a.\, e_F \mid e_F\, e_F \mid e_F\, \sigma$$

$$\boxed{\Gamma \vdash^F e_F : \sigma \rightsquigarrow e'}$$

$$\frac{x : \mu \in \Gamma}{\Gamma \vdash^F x : \mu \rightsquigarrow x} \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash^F x : \sigma \rightsquigarrow (x :: \sigma)}$$

$$\frac{\Gamma, x : \tau \vdash^F e_F : \phi \rightsquigarrow e'}{\Gamma \vdash^F \lambda(x :: \tau).\, e_F : \tau \to \phi \rightsquigarrow \lambda x.\, e'} \qquad \frac{\Gamma, x : \sigma \vdash^F e_F : \phi \rightsquigarrow e'}{\Gamma \vdash^F \lambda(x :: \sigma).\, e_F : \sigma \to \phi \rightsquigarrow \lambda(x :: \sigma).\, e'}$$

$$\frac{\Gamma \vdash^F e_F : \phi[a_1, \ldots, a_n] \rightsquigarrow e'}{\Gamma \vdash^F \Lambda a_1 \ldots \Lambda a_n.\, e_F : \forall a_1.\ \ldots \forall a_n.\, \phi \rightsquigarrow (\lfloor e' \rfloor :: \forall a_1 \ldots a_n.\, \phi)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^F e_0 : \sigma \rightsquigarrow e_0' \qquad \sigma_0 = \forall \overline{a}_1.\, \sigma_1 \to \forall \overline{a}_2.\, \sigma_2 \to \ldots \forall \overline{a}_n.\, \sigma_n \to \forall \overline{a}_r\, \overline{a}_p.\, \mu_r \\ \sigma_0 \leqslant^n_{\text{ii}} [\overline{a_1 \mapsto \phi_1}]\sigma_1, [\overline{a_1 \mapsto \phi_1}, \overline{a_2 \mapsto \phi_2}]\sigma_2, \ldots, [\overline{a_1 \mapsto \phi_1}, \ldots, \overline{a_n \mapsto \phi_n}, \overline{a_r \mapsto \phi_r}]\mu_r \\ \Gamma \vdash^F e_1 : [\overline{a_1 \mapsto \phi_1}]\sigma_1 \rightsquigarrow e_1' \quad \ldots \quad \Gamma \vdash^F e_n : [\overline{a_1 \mapsto \phi_1}, \ldots, \overline{a_n \mapsto \phi_n}]\sigma_n \rightsquigarrow e_n'\end{array}}{\Gamma \vdash^F e_0\, \overline{\phi}_1\, e_1\, \overline{\phi}_2\, e_2\, \ldots\, \overline{\phi}_n\, e_n\, \overline{\phi}_r : \forall \overline{a}_r.\, \mu_r \rightsquigarrow \begin{cases} e_0'\, e_1' \ldots e_n' & \text{if } \overline{a}_p \text{ is empty} \\ & \text{and } \overline{\phi}_r \text{ fully mono.} \\ (e_0'\, e_1' \ldots e_n' :: \forall \overline{a}_r.\, \mu_r) & \text{otherwise} \end{cases}}$$

$$\text{where} \begin{cases} \lfloor e :: \sigma \rfloor = \lfloor e \rfloor \\ \lfloor e \rfloor = e & \text{otherwise} \end{cases}$$

Fig. 8. Translation from System F

The main difference between GI and System F is that in the former impredicative instantiation is restricted by guardedness. The presented translation relies on annotations to work around them.

- Following GI we present n-ary application – which in the case of System F also includes type application. The given application rule only works if the guardedness restrictions are satisfied, otherwise an annotation on e_0 needs to be added before applying the rule.
- In GI the result of a non-annotated application always gets a top-level monomorphic type. This is not the case in System F, and thus an annotation may be required to further generalize.
- Completely unrestricted instantiation is only available via annotations. In order to apply this translation, we need to split applications so that guardedness guarantees are always met. Every time we split, we introduce a new annotation guiding the type checking process.

3.6 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 3.3 (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* $\sigma_i = \varphi_i\sigma^*$.

Proof. See Appendix C.1. □

Corollary 3.4. *Let* Γ *be a closed environment. If* $\Gamma \vdash e : \sigma_1$ *and* $\Gamma \vdash e : \sigma_2$*, then there exists a polymorphic type* σ^* *and fully monomorphic substitutions* φ_1 *and* φ_2 *such that* $\sigma_i = \varphi_i\sigma^*$.

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 3.3. The proof of the principal types property, however, is a corollary of other properties of the inference process, which we describe in Section 4.5.

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.

Theorem 3.5. *If* $\Gamma \vdash u : \sigma$ *and* $\Gamma, x : \sigma \vdash e[x] : \phi$*, then* $\Gamma \vdash e[u] : \phi$.

Proof. By induction over the expression $e[x]$. The only interesting case is when x is the head of an application, that is, $e = x\, e_1\, \ldots\, e_n$ and u is also an application, $u = h\, u_1\, \ldots\, u_n$. Note that in that case the assigned types are always top-level monomorphic, so $\sigma = \mu$ and $\phi = \eta$.

The derivations involved look like:

$$\frac{\Gamma \vdash^{\mathsf{h}} h : \sigma_u \qquad \sigma_u \leqslant^m_{\mathfrak{m}} \sigma_1, \ldots, \sigma_m, \mu \qquad \overline{\Gamma \vdash^{\mathsf{arg}} u_i : \sigma_i}}{\Gamma \vdash h\, u_1\, \ldots\, u_m : \mu} \qquad \frac{\Gamma \vdash^{\mathsf{h}} x : \mu \qquad \mu \leqslant^m_{\mathfrak{m}} \phi_1, \ldots, \phi_n, \eta \qquad \overline{\Gamma \vdash^{\mathsf{arg}} e_j : \phi_j}}{\Gamma \vdash x\, e_1\, \ldots\, e_n : \eta}$$

and the question is whether we can derive:

$$\frac{\Gamma \vdash^{\mathsf{h}} h : \sigma_u \qquad \sigma_u \leqslant^{m+n}_{\mathfrak{m}} \sigma_1, \ldots, \sigma_m, \phi_1, \ldots, \phi_n, \eta \qquad \overline{\Gamma \vdash^{\mathsf{arg}} u_i : \sigma_i} \qquad \overline{\Gamma \vdash^{\mathsf{arg}} e_j : \phi_j}}{\Gamma \vdash h\, u_1\, \ldots\, u_m\, e_1\, \ldots\, e_n : \eta}$$

All the premises on this last rule come directly from those in the hypotheses, except for the instantiation $\sigma_u \leqslant^{m+n}_{\mathfrak{m}} \sigma_1, \ldots, \sigma_m, \phi_1, \ldots, \phi_n, \eta$. For the n last components we can reuse the derivation in the second hypothesis. For the first m components, inspection on the rules of guardedness show that any instantiation with m arguments is admissible when $m + n$ are considered. □

Theorem 3.6. *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* f *and* e *such that* $\Gamma \vdash^{\mathsf{h}} f : \sigma_0$*, and* $\sigma_0 \leqslant^0_{\mathfrak{m}} \sigma_1 \to \phi$ *then:*

$$\Gamma \vdash f\ e : \phi \iff \Gamma \vdash app\ f\ e : \phi \iff \Gamma \vdash revapp\ e\ f : \phi$$

The hypothesis $\sigma_0 \leqslant^0_{\mathfrak{m}} \sigma_1 \to \phi$ 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.

Proof. Let us compare the derivations of the first two expressions.

$$\frac{\Gamma \vdash^{\mathsf{h}} f : \sigma_0 \qquad \sigma_0 \leqslant^1_{\mathfrak{m}} \sigma_1, \mu \qquad \Gamma \vdash^{\mathsf{arg}} e : \sigma_1}{\Gamma \vdash f\ e : \mu}$$

$$\frac{\Gamma \vdash^{\mathrm{h}} app : \forall\, a\ b.\, (a \to b) \to a \to b \qquad \dfrac{\Gamma \vdash^{\mathrm{h}} f : \sigma_0 \qquad \sigma_0 \leqslant^0_{\mathrm{m}} \sigma_1 \to \mu}{\Gamma \vdash^{\mathrm{arg}} f : \sigma_1 \to \mu} \qquad \Gamma \vdash^{\mathrm{arg}} e : \sigma_1}{\Gamma \vdash f\ e : \mu} \quad \forall\, a\ b.\, (a \to b) \to a \to b \leqslant^2_{\mathrm{m}} (\sigma_1 \to \mu), \sigma_1, \mu$$

The application of *app* can be instantiated with any type, given that both variables a and b are guarded. The premises coming from f and e are the same, except for the highlighted ones. We know that if $\sigma_0 \leqslant^0_{\mathrm{m}} \sigma_1 \to \phi$, then $\sigma_0 \leqslant^1_{\mathrm{m}} \sigma_1, \phi$, by inspection of the rule relating instantiation and function types. The case for *revapp* is similar. □

4 TYPE INFERENCE USING CONSTRAINTS

In the previous section we described GI from a declarative perspective. But the utility of the system depends critically on whether we can formulate an efficient type inference algorithm for GI, a question that is the focus of this section.

Our approach to type inference is based on [Pottier and Rémy 2005], which works in two steps:

(1) *Constraint generation*: walk over the syntax tree of the source program, generating *typing constraints*. Constraint-generation is a simple algorithm over a large source-language syntax; i.e. there are many simple cases to consider. Constraint generation typically generates many *unification variables* that stand for as-yet-unknown types.
(2) *Constraint solving*: solve the typing constraints generated in step 1. Constraint solving is a complex algorithm that works over a small language of constraints; i.e. there are few cases to consider but constraint solving is subtle. Constraint solving not only determines satisfiability (i.e. whether the program is typeable at all) but also a *substitution* that specifies the type that each unification variable stands for.

An attractive feature of this two-step approach is that it scales well with the complexity of the type system. For example, earlier work dubbed OutsideIn(X) explains how to apply the Pottier and Rémy [2005] approach to a language like Haskell, with type classes, type-level functions, GADTs, and the like [Vytiniotis et al. 2011]. Another advantage is that is supports more sophisticated type-error diagnosis [Heeren et al. 2003; Serrano and Hage 2016; Zhang et al. 2015].

Performing type inference via constraints may carry a performance cost, due to the increased book-keeping. GHC alleviates this by solving "easy" constraints on-the-fly using ordinary unification, rather than emitting them to be solved later. Doing so might threaten the constraint-based error diagnosis; however the on-the-fly solver can easily be disabled when one wants to prioritise error messages over performance.

4.1 Constraints

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

$$head\ (\forall\, a.\, a \to a)\ ids\ Bool\ True$$

That is, we instantiate *head* at type $(\forall\, a.\, a \to a)$, then apply it to *ids*, to produce a result of type $(\forall\ a.\, a \to a)$. Now we must in turn instantiate that type with *Bool* to get a function of type $(Bool \to Bool)$ 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

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

Fig. 9. Extended syntax

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 $((:)\ (\lambda x.\ x)\ ids)$. In System F terms, we are trying to infer the following elaborated program:

$$(:)\ (\forall\, a.\ a \to a)\ (\Lambda\, a.\ \lambda(x : a).\ x)\ ids$$

in which $(:)$ is instantiated at type $(\forall\, a.\ a \to a)$, 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 $\forall\, a.\ a \to a$, 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 9 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 9, 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.

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

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

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

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

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

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

Fig. 10. Constraint generation, part 1

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 in Section 5, we shall impose a distinction between simple constraints – those which the programmer can type in a program – and extended ones – which are internal to the algorithm.

4.2 Constraint generation

Constraint *generation* is described in Figure 10 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 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: annotations, **let** bindings and pattern matching.

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 $\Gamma \vdash^{\mathsf{h}} h : \phi \rightsquigarrow C$, which either looks up a variable in the environment or threads the information to the normal gathering process.

Then the first column of constraints, each of form $\beta_i^{\mathfrak{u}} \leqslant_{\mathfrak{m}}^{n-i} \alpha_{i+1}^{\mathfrak{u}} \to \beta_{i+1}^{\mathfrak{u}}$, successively decompose 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 $\leqslant_{\mathfrak{s}}^{n}$ rather than type equality. Note that what we express declaratively in a single relation $\sigma_0 \leqslant_{\mathfrak{s}}^{n} \sigma_1, \ldots, \sigma_n, \mu$ is expressed as a sequence of constraints

$$\sigma_0 \leqslant_{\mathfrak{s}}^{n} \sigma_1 \to \phi_1 \quad \phi_1 \leqslant_{\mathfrak{s}}^{n-1} \sigma_2 \to \phi_2 \quad \ldots \quad \phi^{n-1} \leqslant_{\mathfrak{s}}^{1} \sigma_n \to \phi_n \quad \phi_n \leqslant_{\mathfrak{s}}^{0} \mu$$

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

The second column of constraints introduces a completely new constraint form, $\mathfrak{g} \preceq \sigma$, 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 $\leqslant_{\mathfrak{u}}^{n}$ in the column of instantiation constraints, with suffix $\mathfrak{u}$, rather than $\mathfrak{m}$ 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 $\overline{v} = \mathsf{fuv}(C)$, and repeatedly applies the solver rules in Figure 11, 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. $\mathsf{Int} \sim \mathsf{Bool}$) 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 11 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 constraint. 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 eqrel removes trivial equality constraints of the form $\sigma \sim \sigma$. Note that two polymorphic types need to be *syntactically equal* (modulo α-equality) to match this rule. 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).

$$\boxed{\mathsf{freshen}^n_{\mathfrak{s}}(\sigma) \implies \langle \overline{v}, \mu \rangle}$$

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

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

$$[\textsc{conj}]\ \frac{C_1\ ;\ \overline{v} \implies C_1'\ ;\ \overline{v}'}{C_1 \wedge C_2\ ;\ \overline{v} \implies C_1' \wedge C_2\ ;\ \overline{v}'}$$

$$[\textsc{forall}]\ \frac{C\ ;\ \overline{v}_{\mathsf{in}} \implies C'\ ;\ \overline{v}'_{\mathsf{in}}}{\forall \overline{a}.\, \exists \overline{v}_{\mathsf{in}}.\, C\ ;\ \overline{v} \implies \forall \overline{a}.\, \exists \overline{v}'_{\mathsf{in}}.\, C'\ ;\ \overline{v}}$$

$$\begin{array}{lrcll}
[\textsc{eqrefl}] & \sigma \sim \sigma\ ;\ \overline{v} & \implies & \top\ ;\ \overline{v} & \\
[\textsc{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} & \\
[\textsc{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} & \\
 & & & & \text{if } \vdash \sigma : \mathfrak{s}, \text{ and } \alpha \notin \mathsf{ftv}(\sigma) \\
[\textsc{eqvar}] & \alpha^{\mathfrak{s}_1} \sim \beta^{\mathfrak{s}_2}\ ;\ \overline{v} & \implies & \beta^{\mathfrak{s}_2} \sim \alpha^{\mathfrak{s}_1}\ ;\ \overline{v} & \text{if } \mathfrak{s}_1 \sqsubset \mathfrak{s}_2 \\
[\textsc{eqfully}] & \alpha^{\mathfrak{m}} \sim \sigma\ ;\ \overline{v} & \implies & \{\beta^{\mathfrak{s}} \sim \gamma^{\mathfrak{m}} \mid \beta^{\mathfrak{s}} \in \mathsf{fuv}(\sigma), \mathfrak{s} \neq \mathfrak{m}\}\ ;\ \overline{v}, \overline{\gamma^{\mathfrak{m}}} & \\
[\textsc{instmono}] & \mu \leqslant^n_{\mathfrak{s}} \eta\ ;\ \overline{v} & \implies & \mu \sim \eta\ ;\ \overline{v} & \\
[\textsc{inst}\forall\textsc{l}] & (\forall \overline{a}.\, \mu) \leqslant^n_{\mathfrak{s}} \eta\ ;\ \overline{v} & \implies & \mu' \sim \eta\ ;\ \overline{v}, \overline{v}' & \\
 & & & & \text{where } \mathsf{freshen}^n_{\mathfrak{s}}(\forall \overline{a}.\, \mu) \implies \langle \overline{v}', \mu' \rangle \\
[\textsc{inst}\forall\textsc{l}] & (\forall \{\overline{v}'\}.\, \overline{C} \Rightarrow \sigma) \preceq \eta\ ;\ \overline{v} & \implies & \overline{C} \wedge \left(\sigma \leqslant^0_{\mathfrak{m}} \eta\right)\ ;\ \overline{v}, \overline{v}' & \\
[\textsc{inst}\forall\textsc{r}] & \mathfrak{g} \preceq (\forall \overline{a}.\, \mu)\ ;\ \overline{v} & \implies & \forall \overline{a}.\, (\mathfrak{g} \preceq \mu)\ ;\ \overline{v} &
\end{array}$$

Fig. 11. Solving rules

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 $\alpha^{\mathfrak{u}}$ and a top-level monomorphic $\beta^{\mathfrak{t}}$, then $\alpha^{\mathfrak{u}}$ should be replaced by $\beta^{\mathfrak{t}}$, and not the other way around. Full monomorphism goes deeper: EQFULLY ensures that if a type σ is equated with a fully monomorphic variable $\alpha^{\mathfrak{m}}$, all the variables in σ turn 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 ⊥. Instead, that constraint is simply stuck, and we can report it at the end.

4.3.2 Instantiation and generalisation constraints. The last set of rules take care of 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 $\mathsf{freshen}^n_{\mathfrak{s}}$, 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$\forall$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 v' into the current constraint.

INST$\forall$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$\forall$R achieves this rather neatly simply by doing skolemisation and pushing the $\mathfrak{g}$ inside; then INST$\forall$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 $\alpha^{\mathfrak{u}} \leqslant^n_{\mathfrak{s}} \mu$ we cannot turn it directly into $\alpha^{\mathfrak{u}} \sim \mu$, because $\alpha^{\mathfrak{u}}$ might be unified later to a polymorphic type and we need instantiation.
(2) Similarly, if we have $(\forall\{\overline{\alpha}\}.\,C \Rightarrow \mu) \leq \alpha^{\mathfrak{u}}$, and $\alpha^{\mathfrak{u}}$ 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 constraint 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. [Comon and Lescanne 1988] (which in our system is expressed by quantification constraints). A complete algorithm to solve this kind of problem is described by Pottier and Rémy [2005]. As a consequence, we have this result:

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 [Vytiniotis et al. 2011]. Furthermore, the approach by Pottier and Rémy [2005] is no longer applicable. With that in mind, Figure 12 presents one extra rule that, applied exhaustively like the others, also solves the problem of unification under a mixed

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

Fig. 12. Solving rules for quantification constraints

$$\boxed{\overline{a}\ ;\ \overline{\alpha}\ ;\ \overline{\beta} \vdash C\ \text{solved}}$$

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

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

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

Fig. 13. Definition of solved set of constraints

prefix, and *does* scale to handle local assumptions. It is also rather simple, and is directly inspired by how GHC handles constraints.

Rule FLOAT simply floats 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 11 and Figure 12 is complete for unification problems under a mixed prefix.*

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_0 : \sigma_0 \rightsquigarrow C_0 \qquad \overline{\alpha}, \beta \text{ fresh} \qquad \begin{array}{l}\text{for each branch } K_i\,\overline{x_i} \to e_i \\ \quad \underline{K_i : \forall \overline{a}\,\overline{b_i}.\,\overline{\sigma_i} \to \mathsf{T}\,\overline{a} \in \Gamma} \\ \quad \Gamma, x_i : [\overline{a \mapsto \alpha^{\mathrm{u}}}]\sigma_i \vdash e_i : \phi_i \rightsquigarrow C_i \\ \quad \overline{v}_i = \mathrm{fuv}(\phi_i, C_i) - \mathrm{fuv}(\Gamma) - \overline{\alpha}\end{array}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\{\overline{K\,\overline{x} \to e}\} : \beta^{\mathrm{u}} \rightsquigarrow C_0 \wedge (\sigma_0 \leqslant^0_{\mathrm{m}} \mathsf{T}\,\overline{\alpha^{\mathrm{u}}}) \wedge \forall \overline{b}_i.\,\exists \overline{v}_i.\,(C_i \wedge \beta^{\mathrm{u}} \sim \phi_i)}\ \textsc{Case}$$

Fig. 14. Constraint generation, part 2

$$\boxed{\widehat{C_{\mathrm{s}}} = \theta}$$

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

Fig. 15. Substitution induced by a solved form

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 13 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 $\overline{v}$ into the unconstrained sets, γ_1 and γ_2 resp.

4.4 let bindings and pattern matching

The generation rules for the rest of the language are given in Figure 14. The rules are rather unsurprising. In rule CASE we need a quantification constraint to introduce new skolem constants, for the existentially quantified type variables $\overline{b}_i$ in the data constructor. We refrain from describing the rules for **do** notation for the sake of conciseness; they do not pose any particular problem.

4.5 Soundness and principality

The presented gathering and solving process satisfies the usual properties of soundness and principality with respect to the declarative specification. In order to state these results, we need the auxiliary notion of substitution induced by a solved form, given in Figure 15.

THEOREM 4.4 (SOUDNESS). *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 $\widehat{C_{\mathrm{s}}}$, then we have $\Gamma \vdash e : \widehat{C_{\mathrm{s}}}(\sigma)$.*

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.*

Polymorphic types	σ, ϕ	::=	$\alpha^{\mathfrak{u}} \mid \forall \overline{a}.\ Q \Rightarrow \mu$	
Simple constraints	Q	::=	$\top \mid Q_1 \wedge Q_2$	
		$\mid$	$\sigma \sim \phi$	Equality
		$\mid$	$\cdots$	Open for extension
		$\mid$	$C\ \sigma_1 \ldots \sigma_n$	Type classes, for example
Extended constraints	C	::=	$Q \mid C_1 \wedge Q_2$	
		$\mid$	$\sigma \leqslant^{n}_{\mathfrak{s}} \mu$	Instantiation
		$\mid$	$\mathfrak{g} \preceq \sigma$	Generalisation
		$\mid$	$\forall \overline{a}.\ \exists \overline{v}.\ (\ Q \supset C)$	Quantification and implication

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

$$\frac{\vdots \qquad \vdots}{\Gamma \vdash (h\, e_1\ \ldots\ e_n :: \forall \overline{b}.\, Q \Rightarrow \eta) : \forall \overline{b}.\, Q \Rightarrow \eta \rightsquigarrow\ \forall \overline{b}.\, \exists \overline{v}.\, (\ Q \supset C \wedge \ldots)}\ \textsc{AnnApp}$$

$$\frac{\Gamma \vdash e_0 : \sigma_0 \rightsquigarrow C_0 \qquad \overline{\alpha}, \beta \text{ fresh} \qquad \begin{array}{l} \text{for each branch } K_i\ \overline{x_i} \to e_i \\ \quad K_i : \forall \overline{a}\, \overline{b_i}.\ Q_i \Rightarrow \overline{\sigma_i} \to \mathsf{T}\, \overline{a} \in \Gamma \\ \quad \Gamma, x_i : \overline{[a \mapsto \alpha^{\mathfrak{u}}]\sigma_i} \vdash e_i : \phi_i \rightsquigarrow C_i \\ \quad \overline{v_i} = \mathrm{fuv}(\phi_i, C_i) - \mathrm{fuv}(\Gamma) - \overline{\alpha} \end{array}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\{\overline{K\ \overline{x} \to e}\} : \beta^{\mathfrak{u}} \rightsquigarrow C_0 \wedge (\sigma_0 \leqslant^{0}_{\mathfrak{m}} \mathsf{T}\, \overline{\alpha^{\mathfrak{u}}}) \wedge \forall \overline{b_i}.\, \exists \overline{v_i}.\, (\ Q_i \supset C_i \wedge \beta^{\mathfrak{u}} \sim \phi_i)}\ \textsc{Case}$$

Fig. 16. Extensions for integration with other constraints

The proofs of these results are given in Appendix C.2.

4.6 Implementation

A prototype implementation of the type inference process, optionally including the relaxed conditions described in Section 6, is available at url.anonymized. The expressions in Figure 2 are accepted or rejected as described by the table. The prototype also includes support for Haskell's type classes by extending GI as described in Section 5.

5 INTEGRATION WITH OTHER LANGUAGE FEATURES

One of the main advantages of organising a type checker around the concept of constraints is that extensions to the type system can be accommodated for quite easily. Indeed, this is one of the main advantages of our system over existing work. In this section we describe how to deal with other forms of constraint beyond the standard equality constraints we now support.

For example, Haskell supports *type classes*, which restrict the scope of a polymorphic abstraction to a subset of types. The archetypal example is *Eq*, which describes the types with support for decidable equality. Such a type class constraint is visible in the type of the equality operator $(\equiv) :: \forall\, a.\, Eq\ a \Rightarrow a \to a \to Bool$. Similarly, languages like OCaml and PureScript feature *row* (or *record*) *types* like $\{\, x :: Point, y :: Point \,\}$ in addition to usual ADTs.

The good news is that a constraint-based formulation of typing makes it easy to cope with new concepts if they can also be described in terms of constraints. Several examples in the literature, like HM(X) [Pottier and Rémy 2005] and OutsideIn(X) [Vytiniotis et al. 2011], are actually frameworks which can be parametrized by different constraint systems, hence the X in their names.

The modifications needed to accommodate the new kinds of constraints in GI are given in Figure 16. First, we split up the syntax of constraints into so-called simple and extended constraints. The former consists of constraints that can be used by programmers in their programs, while the second category consists of additional constraints that are internal to the solver. Syntax for any new kind of constraints can be added to the syntax of simple constraints Q; we have in fact done so for the specific example of type classes. We also modify the syntax of polymorphic types so that they may contain a number of simple constraints.

Now, suppose that we need to check that $\forall\, a.\, Ord\ a \Rightarrow a \to Bool$ is an instance of $\forall\, a.\, Eq\ a \Rightarrow a \to Bool$, in other words $(\forall\{\alpha\}.\, Eq\ \alpha \Rightarrow \alpha \to Bool) \preceq (\forall\, a.\, Ord\ a \Rightarrow a \to Bool)$. During this process we are allowed to *assume* that $Ord\ a$ holds in order to discharge $Eq\ \alpha$. To be able to store this information in our extended constraints, we modify the syntax of quantification constraints to include the assumed information as part of an implication, in this case

$$\forall a.\, (Ord\ a \supset \forall\{\alpha\}.\, Eq\ \alpha \Rightarrow \alpha \to Bool \preceq a \to Bool)$$

In turn, this requires changes to the solving rules (Figure 11 and Figure 12), which will then need to relate two sets of constraints: the *assumptions* and the *wanted* constraints. Whenever we go under an implication, we introduce the constraints that are part of the antecedent as additional assumptions. The solver is allowed to rewrite in both sets, and more importantly, to use an assumed constraint to rewrite a wanted one. The modifications to the rules are very much in line with [Vytiniotis et al. 2011] so we do not discuss it further in the paper; we do provide the modified rules in Appendix A.

Implication constraints are also introduced by the updated rules for constraint gathering. An annotated application – rule AnnApp – may also mention constraints, which are assumed while checking the enclosed expression. Rule Case allows some constraints to be locally valid, which means that the system gains support for Generalized Algebraic Data Types (GADTs).

6 RELAXING GUARDEDNESS

In this section we show a very simple change to the constraint solver that allows it to solve more constraints. The downside is that, while it accepts all programs that satisfy the declarative specification, we do not yet have a simple declarative specification that precisely characterises all the programs it accepts.

We start with an example. Consider the expression (*choose* [ ] *ids*) from Figure 2. It fails to typecheck in GI, for the reason discussed in Section 3.3: Rule 1 says that we cannot instantiate the empty list [ ] polymorphically, forbidding the instance $[\forall\, a.\, a \to a]$ which is required. In this case the guardedness restriction seems to be too restrictive: all the information to infer the correct type seems to be there and we are not violating the "polymorphism is not guessed" policy.

What actually goes wrong in the solver? Here are the constraints generated for (*choose* [ ] *ids*):

$$\begin{array}{ccc} \forall\, a.\, a \to a \to a \leqslant^2_{\mathrm{m}} \alpha^{\mathrm{u}}_1 \to \beta^{\mathrm{u}}_1 & \beta^{\mathrm{u}}_1 \leqslant^2_{\mathrm{m}} \alpha^{\mathrm{u}}_2 \to \beta^{\mathrm{u}}_2 & \beta^{\mathrm{u}}_2 \leqslant^0_{\mathrm{m}} \delta^{\mathrm{t}}_3 \\ \forall\{\delta^{\mathrm{t}}_1\}.\, \forall\, a.\, [a] \leqslant^0_{\mathrm{m}} \delta^{\mathrm{t}}_1 \Rightarrow \delta^{\mathrm{t}}_1 \preceq \alpha^{\mathrm{u}}_1 & \forall\{\delta^{\mathrm{t}}_2\}.\, [\forall\, a.\, a \to a] \leqslant^0_{\mathrm{m}} \delta^{\mathrm{t}}_2 \Rightarrow \delta^{\mathrm{t}}_2 \preceq \alpha^{\mathrm{u}}_2 & \end{array}$$

From rule App we get one instantiation constraint and one generalisation constraint for each argument, plus one instantiation constraint for the result. The first instantiation is rewritten (rule InstAL) to $\epsilon^{\mathrm{t}} \to \epsilon^{\mathrm{t}} \to \epsilon^{\mathrm{t}} \leqslant^2_{\mathrm{s}} \alpha^{\mathrm{u}}_1 \to \beta^{\mathrm{u}}_1$ where ϵ^{t} is fresh. ϵ is chosen to be top-level monomorphic since it appears in at least one given argument, but never under a constructor. This constraint in combination with the rest of the first row leads to a series of equalities $\alpha^{\mathrm{u}}_1 \sim \alpha^{\mathrm{u}}_2 \sim \delta^{\mathrm{t}}_3 \sim \epsilon^{\mathrm{t}}$.

We replace α^{u}_1, α^{u}_2, and δ^{t}_1 by the more restrictive ϵ^{t} (rules EqVar and EqSubst). Now we can unwrap the first generalisation constraint (using InstAL), because ϵ^{t} is guaranteed to be top-level

[CONJ]		As before
[FORALL]		As before
[FLOAT]		As before
[EQREFL]		As before
[EQMONO]		As before
[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 $\not\vdash \sigma : \mathfrak{s}$, and $\alpha \notin \mathrm{ftv}(\sigma)$

Fig. 17. Solving rules for phase 2 of relaxed GI

monomorphic. That gives the constraint $(\forall a.a \to a) \leqslant^0_{\mathfrak{m}} \epsilon^{\mathfrak{t}}$, which we can instantiate (INST$\forall$L) to get $\epsilon^{\mathfrak{t}} \sim [\nu^{\mathfrak{m}}]$. The key point here is that $\nu^{\mathfrak{m}}$ is chosen to be *fully* monomorphic because the type of [ ] has no arguments. The final step is to unwrap the second generalisation constraint to get $[\forall\, a.\, a \to a] \leqslant^0_{\mathfrak{m}} [\nu^{\mathfrak{m}}]$, which quickly leads to the residual constraint $\nu^{\mathfrak{m}} \sim \forall\, a.\, a \to a$.

And at this point we are stuck! A monomorphic variable is equated with a quantified type, which is signaled as an error by the solver. But *why* is it an error? Remember, we are trying to find a substitution for the unification variables that solves the entire constraint. Well, making $\nu^{\mathfrak{m}}$ stand for $\forall\, a.\, a \to a$ will certainly solve this one! The reason for the sortedness constraints was to guide instantiation and generalisation. But since there are no instantiation or generalisation constraints, the presence of sorts only stops us from making progress. So at this point we simply ignore the sortedness restriction in EQSUBST and continue solving.

For example, suppose that at the moment that no further rules from Figure 11 can be applied, we are left with the following constraints:

$$C = \alpha^{\mathfrak{m}} \sim \forall\, a.\, a \to a \wedge \beta^{\mathfrak{t}} \sim [\alpha^{\mathfrak{m}}]$$

Note that this is not a solved form, because the equality over $\alpha^{\mathfrak{m}}$ is ill-sorted, and the equality over $\beta^{\mathfrak{t}}$ mentions another unification variable, $\alpha^{\mathfrak{m}}$, for which a further equality holds. Now we start phase 2 of the solver, using only the rules in Figure 17. Apart from the structural rules CONJ, FORALL and FLOAT, in this second phase we only handle equality constraints, *but now without respecting the sorts*. In other words, the $\vdash \sigma : \mathfrak{s}$ condition is dropped from the updated rule EQSUBST, and the old rules EQVAR and EQFULLY – which were responsible for propagating sort restrictions – are gone. Applied to our set of constraints C, running through phase 2 leaves us with the following:

$$\alpha^{\mathfrak{m}} \sim \forall\, a.\, a \to a \wedge \beta^{\mathfrak{t}} \sim [\forall\, a.\, a \to a]\,.$$

If we disregard the sortedness restrictions imposed by guardedness, these constraints are in solved form, and the program is accepted.

A reasonable question to ask is: why do we need phase 2 at all? The reason is that until we have performed substitutions exhaustively, we cannot be sure that we have a solution free of occurs-check errors. This happens, for example, if in the above example we replace the first constraint with $\alpha^{\mathfrak{m}} \sim \forall a.a \to \beta^{\mathfrak{t}} \to a$.

We conclude this section by noting that we have also experimented with other, yet more powerful extensions of the solver, some involving iteration. It remains future work to decide their cost/benefit trade-off, and to seek better declarative specifications. The swamp is calling!

7 RELATED WORK

Early work has showed that full type inference for System F is an undecidable problem [Wells 1993] – partial type inference with known positions of generalisation but unknown instantiations can be reduced to higher-order unification [Pfenning 1988]. It is also folklore that simple specifications of

System F lack principal types, making thus modular type inference and the additions of ML-style let-bindings impossible.

Type inference in the presence of higher-rank types but where instantiation of quantified variables is restricted to monomorphic types is a problem that has received successful and practical solutions [Dunfield and Krishnaswami 2013; Peyton Jones et al. 2007; Rémy 2005]. These systems exploit type annotations and a mixture of bidirectional propagation and unification and enjoy fairly simple declarative specifications while algorithmically being modest extensions of Algorithm W. Because of predicative instantiation they are typically more expressive than a predicative restriction of System F and type programs in Fη, a variant of System F with implicit η-conversions. Notably [Rémy 2005] present a modification that does allow impredicative – but explicit, based on annotations and quantified constraints – instantiation.

On the other hand no satisfactory solution has been presented to-date that tackles impredicativity with a good benefit-to-weight ratio. The MLF work [Botlan and Rémy 2003, 2009; Rémy and Yakobowski 2008] presents a declarative specification and type inference algorithm for an (impredicative) extension of System F that relies on extending quantification with instance bounds. The resulting system is powerful, but also quite complex to implement; in return we get back principal types. Stemming from this work there have been several attempts to simplify the specification and restrict the user-facing part of MLF to System F types, while still relying on the same powerful unification algorithm. For example FPH [Vytiniotis et al. 2008] exposes an extra "box" structure around inferred types (that would be hidden under a constraint in the context in MLF). Flexible types [Leijen 2009] accept quantification over instantiation constraints but not rigid equality constraints, thanks to a slightly different treatment of type annotations than MLF. Implementing any of these systems in a working compiler is a significant undertaking, and so is the integration with other forms of polymorphism such as type classes [Leijen and Löh 2005].

There have been also proposals of systems with simpler algorithms than MLF and hence more likely to be incorporated in existing type inference engines. Boxy Types [Vytiniotis et al. 2006] has been an early – largely unsuccessful due to its specification complexity – attempt to push bidirectional type inference for higher-rank types to do a small amount of impredicative instantiation. HMF [Leijen 2008] on the other hand, stems from a generalisation of the Hindley-Milner algorithm, and keeps some of its nice properties such as principal types but imposes side conditions which range over all possible derivations, making it more difficult to grasp why a program fails to type check. QML [Russo and Vytiniotis 2009], inspired by earlier work on boxed polymorphism [O'Toole and Gifford 1989; Rémy 1994], proposes the introduction of a non-implicitly-instantiable universal quantifier. In QML, let-bindings can still be generalized a-la Hindley-Milner and implicitly instantiated. In addition first-class polymorphic values can be passed and returned, but can only be instantiated and constructed explicitly, through annotations. The presence of two forms of quantifiers as well as a rather inflexible form of annotation hindered the practicality of the system. Recent work [Eisenberg et al. 2016] also proposes the distinction between an implicitly instantiable and an explicitly instantiable universal quantifier; explicit type applications can be impredicative, whereas implicit ones have to be monomorphic.

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 [Botlan and Rémy 2003], HMF [Leijen 2008], FPH [Vytiniotis et al. 2008], HML [Leijen 2009]. 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 the programs that do not require an implicit η-expansion ($k\ h\ lst$) or the use of a polymorphic function argument at two types $\lambda f.\,(f\ 1, f\ True)$. 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. The vanilla (non-relaxed) GI for which a declarative specification exists 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 needs to be instantiated to a function 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 pretty much recovers some of the lost expressivity, and is by construction as expressive as the vanilla GI, but we have yet to determine whether it admits a simple declarative specification.

8 CONCLUSION

In this paper we have introduced the notion of guarded impredicativity, based on the key idea that in a function application, type variables that occur under a type constructor can be safely instantiated impredicatively. As Figure 2 shows, our system is somewhat less powerful than the most expressive impredicative type systems that can be found in the literature (HMF, HML, FPH and MLF). However, our system has the combined advantage of easy integration with existing constraint solving frameworks such as GHC's OutsideIn(X) and a concise declarative specification making it easy for programmers to predict whether their program will compile. We have given a sound and complete inference algorithm, and proven a principal type property for our system. A prototype implementation, including integration with Haskell's type classes, is available.

An obvious candidate for future work is to integrate our type system into GHC proper. Although it will certainly involve some work, e.g., the existing higher-ranked type system now implemented in GHC can be removed, our prototype was constructed in a way that is consistent with the architecture of the current OutsideIn(X) implementation. We do not expect big complications to arise. Because we are then dealing with a single integrated contraint solving framework in GHC, we can investigate the use of heuristics for type error improving diagnosis and domain-specific type error diagnosis within the context of GHC [Hage and Heeren 2007; Serrano and Hage 2016].

As we hinted at the end of Section 6, we have already looked at yet more powerful extensions of the solver, some involving iteration, for which we need to come up with a suitable declarative specification.

As the reader may have noticed, some expressions need to η-expanded in order to type check. This a direct consequence of the fact that all our type constructors are invariant. Seeing how far we remove the need for η-expansion is yet another interesting topic for further research.

REFERENCES

Lennart Augustsson. 2011. Impredicative polymorphism, a use case. (2011). http://augustss.blogspot.nl/2011/07/impredicative-polymorphism-use-case-in.html.

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. DOI: http://dx.doi.org/10.1145/944705.944709

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

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

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. DOI: http://dx.doi.org/10.1145/2500365.2500582

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. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_10

Jurriaan Hage and Bastiaan Heeren. 2007. Heuristics for type error discovery and recovery. In *Implementation of Functional Languages – IFL 2006*, Z. Horváth, V. Zsók, and A. Butterfield (Eds.), Vol. 4449. Springer Verlag, Heidelberg, 199 – 216.

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

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.

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism, See [Hook and Thiemann 2008], 283–294. DOI: http://dx.doi.org/10.1145/1411204.1411245

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. DOI: http://dx.doi.org/10.1145/1480881.1480891

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. DOI: http://dx.doi.org/10.1145/1086365.1086385

Simon Marlow and others. 2010. Haskell 2010 Language Report. (2010). https://www.haskell.org/onlinereport/haskell2010/.

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

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. DOI: http://dx.doi.org/10.1145/73141.74836

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.

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. DOI: http://dx.doi.org/10.1145/62678.62697

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/

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

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. DOI: http://dx.doi.org/10.1145/1086365.1086383

Didier Rémy and Boris Yakobowski. 2008. From ML to ML^F: graphic type constraints with efficient type inference, See [Hook and Thiemann 2008], 63–74. DOI: http://dx.doi.org/10.1145/1411204.1411216

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. DOI: http://dx.doi.org/10.1145/1596627.1596630

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. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_26

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. DOI: http://dx.doi.org/10.1017/S0956796811000098

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. DOI: http://dx.doi.org/10.1145/1159803.1159838

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell, See [Hook and Thiemann 2008], 295–306. DOI: http://dx.doi.org/10.1145/1411204.1411246

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

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. DOI: http://dx.doi.org/10.1145/2737924.2738009