A Quick Look at Impredicativity

ALEJANDRO SERRANO, 47 Degrees, Spain and Utrecht University, The Netherlands
JURRIAAN HAGE, Utrecht University, The Netherlands
SIMON PEYTON JONES, Microsoft Research, United Kingdom
DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom

Type inference for parametric polymorphism is wildly successful, but has always suffered from an embarrassing flaw: polymorphic types are themselves not first class. We present Quick Look, a practical, implemented, and deployable design for impredicative type inference. To demonstrate our claims, we have modified GHC, a production-quality Haskell compiler, to support impredicativity. The changes required are modest, localised, and are fully compatible with GHC's myriad other type system extensions.

This version has some minor typos fixed, relative to the published ICFP'20 version.

CCS Concepts: • **Theory of computation** → **Type structures**.

Additional Key Words and Phrases: Type systems, impredicative polymorphism, constraint-based inference

ACM Reference Format:
Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (August 2020), 33 pages. https://doi.org/10.1145/3408971

1 INTRODUCTION

Parametric polymorphism backed by Damas-Milner type inference was first introduced in ML [Milner 1978], and has been enormously influential and widely used. But despite this impact, it has always suffered from an embarrassing shortcoming: *Damas-Milner type inference, and its many variants, cannot instantiate a type variable with a polymorphic type*; in the jargon, the system is *predicative*.

Alas, predicativity makes polymorphism a second-class feature of the type system. The type $\forall a.[a] \rightarrow [a]$ is fine (it is the type of the list reverse function), but the type $[\forall a.a \rightarrow a]$ is not, because a $\forall$ is not allowed inside a list. So $\forall$-types are not first class: they can appear in some places but not others. Much of the time that does not matter, but sometimes it matters a lot; and, tantalisingly, it is often "obvious" to the programmer what the desired impredicative instantiation should be (Section 2).

Thus motivated, a long succession of papers have tackled the problem of type inference for impredicativity [Botlan and Rémy 2003; Leijen 2008, 2009; Serrano et al. 2018; Vytiniotis et al. 2006, 2008]. None has succeeded in producing a system that is simultaneously expressive enough to be useful, simple enough to support robust programmer intuition, compatible with a myriad other

Authors' addresses: Alejandro Serrano, alejandro.serrano@47deg.com, 47 Degrees, San Fernando, Spain, A.SerranoMena@uu.nl, Utrecht University, Utrecht, The Netherlands; Jurriaan Hage, J.Hage@uu.nl, Utrecht University, Utrecht, The Netherlands; Simon Peyton Jones, simonpj@microsoft.com, Microsoft Research, Cambridge, United Kingdom; Dimitrios Vytiniotis, dvytin@google.com, DeepMind, London, United Kingdom.

2475-1421/2020/8-ART89
https://doi.org/10.1145/3408971

type system extensions, and implementable without an invasive rewrite of a type inference engine tailored to predicative type inference.

In Section 3 we introduce Quick Look, a new inference algorithm for impredicativity that, for the first time, (a) handles many "obvious" examples; (b) is expressive enough to handle all of System F; (c) requires no extension to types, constraints, or intermediate representation; and (d) is easy and non-invasive to implement in a production-scale type inference engine – indeed we have done so in GHC. We make the following contributions:

- We formalise a higher-rank baseline system (Section 4), and give the changes required for Quick Look (Section 5). A key property of Quick Look is that it requires only highly localised changes to such a specification. In particular, no new forms of types are required, and programs can be elaborated into a statically typed intermediate language based on System F. Some other approaches, such as MLF [Botlan and Rémy 2003], require substantial changes to the intermediate language, but Quick Look does not.
- We prove a number of theorems about our system, including about which transformations do, and do not, preserve typeability (Section 6).
- We give a type inference algorithm for Quick Look (Section 7). This algorithm is based on the now-standard approach of first *generating typing constraints* and then *solving them* [Pottier and Rémy 2005]. As in the case of the declarative specification, no new forms of types or constraints are needed. Section 7 proves its soundness compared with the declarative specification in Section 5.[1] The implementation is in turn based very closely on this algorithm. The constraint generation judgements in Sections 7 and 8 also appear to be the first formal account of the extremely effective combination of bidirectional type inference [Peyton Jones et al. 2007] with constraint-based type inference [Pottier and Rémy 2005; Vytiniotis et al. 2011],
- Because Quick Look's impact is so localised, it is simple to implement, even in a production compiler. Concretely, the implementation of Quick Look in GHC, a production compiler for Haskell, affected only 1% of GHC's inference engine.
- To better support impredicativity, we propose to abandon contravariance of the function arrow (Section 5.8). There are independent reasons for making this change [Peyton Jones 2019], but it is illuminating to see how it helps impredicativity. We also provide data on its impact (Appendix A).

The paper uses a very small language, to allow us to focus on impredicativity, but Quick Look scales very well to a much larger language. Section 8 and Appendix B give the details for a much richer set of features.

We present our work in the concrete setting of (a tiny subset of) Haskell, but there is nothing Haskell-specific about it. Quick Look could readily be used in any other type inference system. We cover the rich related work in Section 11.

2 MOTIVATION

The lack of impredicativity means that polymorphism is fundamentally second class: *we cannot abstract over polymorphic types.* For example, even something as basic as function composition fails on functions with higher-rank types (types with foralls in them). Suppose

$$f :: (\forall a.[a] \to [a]) \to Int \qquad g :: Bool \to \forall a.[a] \to [a]$$

Then the composition $(f \circ g)$ fails typechecking, despite the obvious compatibility of the types involved, simply because the composition requires instantiating the type of $(\circ)$ with a polytype.

[1]We conjecture that completeness is true as well – we are not aware of any counterexample.

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

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

As another example, Augustsson describes an application [Augustsson 2011] in which it was crucial to have a function $var :: RValue\ a \to IO\ (\forall lr.LR\ lr \Rightarrow lr\ a)$, an IO action that returns a polymorphic value. Yet in Haskell today, this is out of reach; instead you have to define a new *named* type, thus:

newtype $LRType\ a = MkLR\ (\forall lr.LR\ lr \Rightarrow lr\ a)$
$var :: RValue\ a \to IO\ (LRType\ a)$

Every use of *var* must pattern match to unwrap the newtype. We call this approach "boxed impredicativity", because the forall is wrapped in a named "box", here *LRType*. But boxed impredicativity is tiresome at best, and declaring a new type for every polymorphic shape is gruesome.

Why not simply allow first-class polymorphism, so that $[\forall a.a \to a]$ is a valid type? The problem is in *type inference.*[2] Consider the expression (*single id*), where the type of *single* and *id* are given in Figure 1. It is not clear whether to instantiate p with $\forall a.a \to a$, or with $Int \to Int$, or some other monomorphic type. Indeed (*single id*) does not even have a most general (principal) type: it has two incomparable types: $\forall a.[a \to a]$ and $[\forall a.a \to a]$. Losing principal types, especially for such an innocuous program, is a heavy price to pay for first-class polymorphism.

But in many cases there is no such problem. Consider (*head ids*) where, again, the types are given in Figure 1. Now there is no choice: the only possibility is to instantiate p with $\forall a.a \to a$. Our idea, just as in previous work [Serrano et al. 2018], is to exploit that special case. Our overall goals are these:

- *First class polymorphism.* We want forall-types to be first class. A function like list $reverse :: \forall a.[a] \to [a]$ should work as uniformly over $[\forall a.a \to a]$ as it does over $[Int]$ and $[Bool]$, and should do so without type annotations. No mainstream deployed language allows that; and not being able to do so is a fundamental failure of abstraction. Using boxed impredicativity is an anti-modular second best.
- *Predictable type inference:* it should be possible for programmers to acquire a robust mental model of what will typecheck and what will not. Typically they do so through a process of trial and error, but our formalism in Section 5 is specifically designed to enshrine the common-sense idea that when there is clear evidence (through the argument or result type) about how to instantiate a call, type inference should take advantage of it.

[2]Type inference is in fact undecidable for System F [Pfenning 1995; Wells 1993].

- *Minimize type annotations:* "obviously typeable" programs should be typeable without annotation. To substantiate this necessarily-qualitative claim we give numerous examples, especially in Figure 12.
- *Conservative extension of Damas-Milner* and its extensions to type classes, higher rank, etc. That is, existing programs continue to typecheck (Section 6.1).
- *Can express all of System F*, with the use of type annotations (Section 6.1).
- *Localised, in both specification and implementation.* We seek a system that affects only a small part of the specification, and the implementation, of the type system and its inference algorithm. Modern type systems, such as that of Haskell, OCaml, or Scala, are subtle and complicated; anything that requires pervasive changes is unlikely to be implemented.

3 THE QUICK LOOK

Our new approach works as follows:

- Treat applications as a whole: a function applied to a list of arguments. The list of arguments can be empty, in which case the "function" is not necessarily a function: it can be a polymorphic value, such as the empty list $[\,] :: \forall p.[p]$.
- When instantiating the function, take a quick look at the arguments to guide that (possibly impredicative) instantiation.
- If Quick Look produces a definite answer, use it; otherwise instantiate with a monotype (as usual in Hindley-Damas-Milner type inference).

In our example ($head\ ids$), we have to instantiate the type of $head :: \forall p.[p] \to p$. The argument $ids :: [\forall a.a \to a]$ must be compatible with the type $head$ expects, namely $[p]$. So we are forced to instantiate $p := \forall a.a \to a$.

On the other hand for ($single\ \ id$), Quick Look sees that the argument $id :: \forall a.a \to a$ must be compatible with the type *single* expects, namely p. But that does not tell us what p must be: should we instantiate that $\forall a$ or not? So Quick Look produces no advice, and we revert to standard Hindley-Damas-Milner type inference by instantiating p with a monotype $\tau \to \tau$. (Operationally, the inference algorithm will instantiate p with a unification variable.)

Why is ($head\ \ ids$) easier? Because the type variable p in *head*'s type appears *guarded*, under the list type constructor; but not so for *single*. Exploiting this guardedness was the key insight of earlier work [Serrano et al. 2018].

The Quick Look approach scales nicely to handle multiple arguments. For example, consider the expression ($id : ids$), where (:) is Haskell's infix cons operator. How should we instantiate the type of (:) given in Figure 1? Taking a quick look at the first argument, *id*, yields no information; it is like the ($single\ id$) case. But at the second argument, *ids*, it immediately tells us that p must be instantiated with $\forall a.a \to a$. We gain a lot from taking a quick look at *all* the arguments before committing to *any* instantiation.

3.1 Quick Look at the Result

So far we have concentrated on using the *arguments* of a call to guide instantiation, but we can also use the *result* type. Consider this expression, which has a user-written type signature:

$$(single\ id) :: [\forall a.a \to a]$$

When considering how to instantiate *single*, we know that it produces a result of type $[p]$, which must fit the user-specified result type $[\forall a.a \to a]$. So again there is only one possible choice of instantiation, namely $p := \forall a.a \to a$.

This same mechanism works when the "expected" type comes from an enclosing call. Suppose *foo* :: $[\forall a.a \to a] \to Int$, and consider *foo* (*single id*). The context of the call (*single id*) specifies the result type of the call, just as the type signature did before. We need to "push down" the type expected by the context into an expression, but fortunately this ability is already well established in the form of *bidirectional type inference* [Peyton Jones et al. 2007; Pierce and Turner 2000] as Section 4 discusses.

Taking a quick look at the result type is particularly important for lone variables and constants. We can treat those as degenerate forms of call with zero arguments. Its instantiation cannot be informed by a quick look at the arguments, since it has none; but it can benefit from the result type. A ubiquitous example is the empty list [ ] :: $\forall p.[p]$. Consider the task of instantiating [ ] in the context of a call *foo* [ ]. Since *foo* expects an argument of type $[\forall a.a \to a]$, the only way to instantiate [ ] is with $p := \forall a.a \to a$.

Finally, here is a more complicated example. Consider the call ([ ] ++ *ids*), where the types are given in Figure 1. First we decide how to instantiate (++) and, as in the case of *head*, we can discover its instantiation $p := \forall a.a \to a$ from its second argument *ids*. Having made that decision we now typecheck its first argument, [ ], knowing that the result type must be $[\forall a.a \to a]$, and that in turn tells us the instantiation of [ ].

3.2 Richer Arguments

So far the argument of every example call has been a simple variable. But what if it was a list comprehension? A lambda? Another call?

One strength of the Quick Look approach is that we are free to make restrictions without affecting anything fundamental. For example, we could say (brutally) that Quick Look yields no advice for an argument other than a variable. The "no advice" case simply means that we will look for information in other arguments or, if none of them give advice, revert to monomorphic instantiation.

We have found, however, that it is both easy and beneficial to allow nested calls. For example, consider (*id* : (*id* : *ids*)). We can only learn the instantiation of the outer (:) by looking at its second argument (*id* : *ids*), which is a call. It would be a shame if simple call nesting broke type inference.

However, allowing nested calls is (currently) where we stop: if you put a list comprehension as an argument, Quick Look will ignore that argument. Allowing calls seems to be a sweet spot. One could go further, but the cost/benefit trade-off seems much less attractive.

The alert reader will note that Quick Look appears to have complexity quadratic in the depth of function call nesting. In our example (*id* : (*id* : *ids*)) the depth was two, but if there were many elements in the list, each nested call would take a quick look into its argument, with cost linear in the depth of that argument. Happily, our implementation completely avoids this complexity change, by retaining and re-using results of the recursive quick looks (Section 9).

3.3 Uncurried Functions

We have focused on exploiting n-ary calls of curried functions, but Quick Look works equally well on *uncurried* functions. For example, suppose *cons* :: $\forall p.(p, [p]) \to [p]$, and we have the call *cons* (*id*, *ids*). Quick Look only has one argument to consult, namely a nested call to the pair constructor. So again, supporting nested calls is necessary, and it rapidly discovers that the only possible instantiation is $p := \forall a.a \to a$.

3.4 Interim Summary

The new Quick Look phase guides instantiation of a call based on the context of the call: its arguments and expected result type. Quick Look is a modular addition: it guides instantiation at call sites, but the entire inference algorithm is otherwise undisturbed. That is in sharp contrast to

Type constructors		$\ni$	$\mathsf{F}, \mathsf{G}, \mathsf{T}, \ldots$	Includes $(\to)$
Type variables		$\ni$	$a, b, \ldots$	
Term variables		$\ni$	$x, y, f, g, \ldots$	
Instantiation variables		$\ni$	κ, μ, υ	
Polymorphic types	σ, ϕ	$::=$	$\forall a.\sigma \mid \rho$	
Top-level mono. types	ρ	$::=$	$\kappa \mid \tau \mid \mathsf{T}\,\overline{\sigma}$	
Fully mono. types	τ	$::=$	$a \mid \mathsf{T}\,\overline{\tau}$	
Typechecking direction	δ	$::=$	$\Uparrow \mid \Downarrow$	Inference and checking respectively
Application heads	h	$::=$	x	Variable
		$\mid$	$e :: \sigma$	Annotation
		$\mid$	e	(not an application)
Arguments	π	$::=$	$\sigma \mid e$	
Terms / expressions	e	$::=$	$h\,\pi_1 \ldots \pi_n$	Application ($n \geqslant 0$)
		$\mid$	$\lambda x.\,e$	Abstraction
Environments	Γ	$::=$	$\epsilon \mid \Gamma, x{:}\sigma \mid \Gamma, a$	

Mono-substitutions	θ, ψ	$::=$	$[\overline{\alpha := \tau}]$	$\mathsf{fiv}(\sigma)$	Free instantiation vars. of σ
Poly-substitutions	Θ, Ψ	$::=$	$[\overline{\kappa := \sigma}]$	$\mathsf{dom}(\theta)$	Domain of θ
				$\mathsf{rng}(\theta)$	Range of θ
				$\mathsf{valargs}(\overline{\pi})$	Value arguments in $\overline{\pi}$

Fig. 2. Syntax

earlier approaches, which have a pervasive effect throughout type inference. It seems plausible, therefore, that the Quick Look approach would work equally well in other languages with very different type inference engines.

4 BIDIRECTIONAL, HIGHER-RANK INFERENCE

We begin our formalisation by giving a solid baseline, closely based on *Practical type inference for arbitrary-rank types* [Peyton Jones et al. 2007], which we abbreviate PTIAT. We simplify PTIAT by omitting the so called "deep skolemisation" and instantiation, and covariance and contravariance in function arrows, a choice we discuss in Section 5.8. We handle function application in an unusual way, one that will extend nicely for Quick Look, and we add *visible type application* [Eisenberg et al. 2016].

4.1 Syntax

The syntax of our language is given in Figure 2.

Types. The syntax of types is unsurprising. Type constructors T include the function arrow $(\to)$, although we usually write it infix. So $(\tau_1 \to \tau_2)$ is syntactic sugar for $((\to)\ \tau_1\ \tau_2)$. A top-level monomorphic type, ρ, has no *top-level* foralls, but may contain nested foralls; while a fully monomorphic type, or *monotype*, τ, has no foralls anywhere. Notice that in a polytype σ the foralls can occur arbitrarily nested, including to the left or right of a function arrow. However, a *top-level monomorphic type* ρ has no foralls at the top.

Terms. In order to focus on impredicativity, we restrict ourselves to a tiny term language: just the lambda calculus plus type annotations. We do not even support **let** or **case**. However, nothing

$$\boxed{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma}$$

$$\frac{\Gamma, \overline{a} \vdash_{\Downarrow} e : \rho}{\Gamma \vdash^{\forall}_{\Downarrow} e : \forall \overline{a}.\, \rho}\ \textsc{gen}$$

$$\boxed{\Gamma \vdash_{\Uparrow} e : \rho}\ \boxed{\Gamma \vdash_{\Downarrow} e : \rho}$$

$$\frac{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma \,;\, \overline{\pi} \leadsto \overline{\phi} \,;\, \rho_r \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \qquad \mathsf{dom}(\theta) = \mathsf{fiv}(\overline{\phi}, \rho_r) \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi_i} \qquad \rho = \theta\rho_r}{\Gamma \vdash_{\delta} h\,\overline{\pi} : \rho}\ \textsc{app-}\delta$$

$$\frac{\Gamma, x{:}\tau \vdash_{\Uparrow} e : \rho}{\Gamma \vdash_{\Uparrow} \lambda x.\, e : \tau \to \rho}\ \textsc{abs-}\Uparrow \qquad \frac{\Gamma, x{:}\sigma_a \vdash^{\forall}_{\Downarrow} e : \sigma_r}{\Gamma \vdash_{\Downarrow} \lambda x.\, e : \sigma_a \to \sigma_r}\ \textsc{abs-}\Downarrow$$

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} x : \sigma}\ \textsc{h-var} \qquad \frac{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} (e :: \sigma) : \sigma}\ \textsc{h-ann} \qquad \frac{\Gamma \vdash_{\Uparrow} e : \rho}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \rho}\ \textsc{h-infer}$$

Fig. 3. Base type system: expressions

essential is thereby omitted. A major feature of Quick Look is that it is completely localised to typing applications. It is fully compatible with, and leaves entirely unaffected, all other aspects of the type system, including ML-style **let**-generalisation, pattern matching, GADTs, type families, type classes, existentials, and the like (Section 5.9).

Similar to other works on type inference [Dunfield and Krishnaswami 2019; Leijen 2008; Vytiniotis et al. 2006] our syntax uses n-ary application. The term $(h\,\pi_1 \ldots \pi_n)$ applies a *head*, h, to a sequence of zero or more arguments $\pi_1 \ldots \pi_n$. The head can be a variable x, an expression with a type annotation $(e :: \sigma)$, or an expression e other than an application. The intuition is that we want to use information from the arguments to inform instantiation of the function's polymorphic variables. In fact, GHC's implementation *already* treats application as an n-ary operation to improve error messages. Note also that a lone variable x is a valid expression e; it is just an n-ary application with no arguments.

An argument π is either a *type argument* σ or a *value argument* e. Type arguments allow the programmer to *explicitly* instantiate the quantified variables of the function (Section 4.4).

4.2 Bidirectional Typing Rules

The typing rules for our language are given in Figures 3 and 4. Following PTIAT, to support higher rank types the typing judgment for terms is *bidirectional*, with two forms: one for checking and one for inference.

$$\Gamma \vdash_{\Downarrow} e : \rho \qquad\qquad \Gamma \vdash_{\Uparrow} e : \rho$$

The first should be read *"in type environment* Γ*, check that the term* e *has type* ρ*"*. The second should be read *"in type environment* Γ*, the term* e *has inferred type* ρ*"*. Notice that in both cases the type ρ

$$\boxed{\Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \rightsquigarrow \overline{\phi} \;;\; \rho_r}$$

$$\frac{\Gamma \vdash^{\mathsf{i}} \sigma \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}{\Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \rightsquigarrow \overline{\phi} \;;\; \rho_r}\ \textsc{inst}$$

$$\boxed{\begin{array}{c}\Gamma \vdash^{\mathsf{i}} \sigma \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r \\ \text{Invariants: } \overline{\phi} \text{ and } \rho_r \text{ are fixed points of } \Theta\text{; length}(\overline{\phi}) = \text{length}(\text{valargs}(\overline{\pi}))\end{array}}$$

$$\frac{}{\Gamma \vdash^{\mathsf{i}} \rho_r \;;\; \epsilon \rightsquigarrow \emptyset \;;\; \epsilon \;;\; \rho_r}\ \textsc{iresult}$$

$$\frac{\overline{\pi} \neq \sigma,\overline{\pi'} \quad \kappa \text{ fresh} \quad \Gamma \vdash^{\mathsf{i}} [a := \kappa]\rho \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}{\Gamma \vdash^{\mathsf{i}} \forall a.\rho \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}\ \textsc{iall}$$

$$\frac{\Gamma \vdash^{\mathsf{i}} \rho[a := \sigma] \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}{\Gamma \vdash^{\mathsf{i}} \forall a.\rho \;;\; \sigma,\overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}\ \textsc{ityarg}$$

$$\frac{\Gamma \vdash^{\mathsf{i}} \sigma_2 \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}{\Gamma \vdash^{\mathsf{i}} (\sigma_1 \to \sigma_2) \;;\; e,\overline{\pi} \rightsquigarrow \Theta \;;\; \Theta\sigma_1,\overline{\phi} \;;\; \rho_r}\ \textsc{iarg}$$

$$\frac{\mu, \upsilon \text{ fresh} \quad \Theta_1 = [\kappa := (\mu \to \upsilon)] \quad \Gamma \vdash^{\mathsf{i}} (\mu \to \upsilon) \;;\; e,\overline{\pi} \rightsquigarrow \Theta_2 \;;\; \overline{\phi} \;;\; \rho_r}{\Gamma \vdash^{\mathsf{i}} \kappa \;;\; e,\overline{\pi} \rightsquigarrow \Theta_2 \circ \Theta_1 \;;\; \overline{\phi} \;;\; \rho_r}\ \textsc{ivar}$$

Fig. 4. Base instantiation

has no top-level quantifiers, but for checking ρ is considered as an *input* while for inference it is an *output*. When a rule has $\vdash_\delta$ in its conclusion, it is shorthand for two rules, one for $\vdash_\Uparrow$ and one for $\vdash_\Downarrow$.

For example, rule ABS-$\Uparrow$ deals with a lambda ($\lambda x.e$) in inference mode. The premise extends the environment Γ with a binding $x : \tau$, for some monotype τ, and infers the type of the body e, returning its type ρ. Then the conclusion says that the type of the whole lambda is $\tau \to \rho$. Note that in inference mode the lambda-bound variable must have a monotype. A term like $\lambda x.\ (x\ \mathit{True}, x\ 3)$ is ill-typed in inference mode, because x (being monomorphic) cannot be applied both to a Boolean and an integer. As is conventional, the type τ appears "out of thin air". When constructing a typing derivation we are free to use any τ, but of course only a suitable choice leads to a valid derivation.

Rule ABS-$\Downarrow$ handles a lambda in checking mode. The type being pushed down must be a function type $\sigma_a \to \sigma_r$; we just extend the environment with $x : \sigma_a$ and check the body. Note that in checking mode the lambda-bound variable *can* have a polytype, so the lambda term in the previous paragraph *is* typeable. Notice that in ABS-$\Downarrow$ the return type σ_r may have top-level quantifiers. The judgment $\vdash^{\forall}_{\Downarrow} e : \sigma$, in Figure 3, deals with this case by adding the quantifiers to Γ before checking the expression against ρ.

The motivation for bidirectionality is that in checking mode we may push down a polytype, and thereby (as we have seen in ABS-$\Downarrow$) allow a lambda-bound variable to have a polytype. But how do we first *invoke* the checking judgment in the first place? One occasion is in rule H-ANN, where we have an explicit, user-written type signature. The second main occasion is in a function application, where we push the type expected by the function into the argument, as we show next.

4.3 Applications and Instantiation

A function application with n arguments (including $n = 0$) is dealt with by rule APP-δ, whose premises perform these five steps:

(1) Infer the (polymorphic) type σ of the function h, using $\vdash^{\mathsf{h}}_{\Uparrow}$. Usually the function is a variable x, and in that case we simply look up x in the environment Γ (rule H-VAR in Figure 3).
(2) Instantiate h's type σ with fresh *instantiation variables*, $\kappa, \mu, \ldots$, guided by the arguments $\overline{\pi}$ to which it is applied, using the judgment $\vdash^{\mathsf{inst}}$. This judgement returns: a type ϕ_i for each of the value arguments in $\overline{\pi}$; and the top-level-monomorphic result type ρ_r of the call.
(3) Conjure up a "magic substitution" θ that maps each of the free instantiation variables in $\overline{\phi}$ and ρ_r to a monotype. Just like the τ in ABS-$\Uparrow$, this θ comes "out of thin air".
(4) Check that each value argument e_i has the expected type $\theta\phi_i$. Note that $\theta\phi_i$ can be an arbitrary polytype, which is pushed into the argument, using the checking judgment $\vdash^{\forall}_{\Downarrow}$. Using the function type to specify the type of each argument is the essence of PTIAT.
(5) Checks that the result type of the call, $\theta\phi_r$ fits the expected type ρ; that is $\rho = \theta\phi_r$

Notice that instantiation variables have a very short, local life: they are born in step (2), and have have completely disappeared by the end of step (3). Instantiation variables never appear in Γ. You may wonder why we did not simply instantiate with arbitrary monotypes in step (2), and dispense with instantiation variables, and with θ. That would be simpler, but dividing the process in two will allow us to modify step (2) to perform Quick Look.

The instantiation judgment, shown in Figure 4, has the form

$$\Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \rightsquigarrow \overline{\phi} \;;\; \rho_r$$

It implements step (2) by instantiating σ, guided by the arguments $\pi_1 \ldots \pi_n$. The type σ and arguments $\overline{\pi}$ should be considered inputs; the argument types $\overline{\phi}$, and result type ρ_r are outputs. The returned arguments $\overline{\phi}$ correspond 1-1 with the value arguments of $\overline{\pi}$. For example,

$$\Gamma \vdash^{\mathsf{inst}} (\forall ab.a \to b \to b) \;;\; \mathit{True} \rightsquigarrow \kappa \;;\; (\upsilon \to \upsilon)$$

The environment Γ (an input) is entirely unused, and in $\overline{\pi}$ the value-argument terms e are unused. Both become important later for Quick Look. Moreover, as you can see from INST in Figure 4, $\vdash^{\mathsf{inst}}$ is merely a wrapper around $\vdash^{\mathsf{i}}$, the workhorse for instantiation: $\vdash^{\mathsf{inst}}$ calls $\vdash^{\mathsf{i}}$ and returns all its results except the substitution Θ. We discuss Θ when we get to rule IVAR.

First though, look at the easy rules for $\vdash^{\mathsf{i}}$, IALL, IARG, and IRESULT. IALL instantiates a leading $\forall$; IARG decomposes a function arrow; and, when the argument list is empty, IRESULT returns the result type ρ_r. Note that the rules deal correctly with function types that have a forall nested to the right of an arrow, e.g. $f :: \mathit{Int} \to \forall a.[a] \to [a]$. For example,

$$\Gamma \vdash^{\mathsf{i}} (\mathit{Int} \to \forall a.[a] \to [a]) \;;\; 3, x \rightsquigarrow \emptyset \;;\; \mathit{Int}, [\kappa] \;;\; [\kappa]$$

We discuss ITYARG in Section 4.4. That leaves IVAR which deals with the case that the function type ends in a type variable, but there is another argument to come. For example, consider $(\mathit{id}\ x\ 3)$. We instantiate id with κ, so $(\mathit{id}\ x)$ has type κ; that appears applied to 3, so we learn that κ must be $\mu \to \upsilon$. We express that knowledge with a little substitution $\Theta_1 = [\kappa := (\mu \to \upsilon)]$, and we return that substitution, composed with Θ_2 which comes back from the recursive call.

$$\Gamma \vdash^{\mathsf{i}} (\forall a.a \to a) \;;\; x, 3 \rightsquigarrow [\kappa := (\mu \to \upsilon)] \;;\; (\mu \to \upsilon), \mu \;;\; \upsilon$$

We must return the substitution Θ so that we can apply it to "earlier" arguments in rule IARG; hence $\Theta\sigma_1$ in the conclusion of that rule. To extend the example, suppose our call was $(id\ x\ 3\ 4)$, where $(id\ x)$ is applied not just to one, but two arguments. Then we have

$$\Gamma \vdash^{i} (\forall a.a \to a)\ ;\ x, 3, 4 \rightsquigarrow [\kappa := (\mu \to \kappa_1), \kappa_1 := (\nu \to \kappa_2)]\ ;\ (\mu \to \nu \to \kappa_2), \mu, \nu\ ;\ \kappa_2$$

Finally $\vdash^{\mathsf{inst}}$ discards the substitution returned by $\vdash^{i}$; it was needed only by IARG. This plumbing of the substitution is a little tiresome, but nothing very deep is happening.

4.4 Visible Type Application

The $\vdash^{\mathsf{inst}}$ judgement also implements visible type application (VTA) [Eisenberg et al. 2016], a popular extension offered by GHC. The programmer can use VTA to *explicitly* instantiate a function call. For example, if $xs :: [Int]$ we could say either $(head\ xs)$ or, using VTA, $(head\ @Int\ xs)$.

Adding VTA has an immediate payoff for impredicativity: an explicit type argument can be a *polytype*, thus allowing explicit impredicative instantiation of any call. This is not particularly convenient for the programmer – the glory of Damas-Milner is that instantiation is silent – but it provides a fall-back that handles all of System F.

More precisely, rule ITYARG (Figure 4) deals with a visible type argument, by using it to instantiate the forall[3]. The argument is a polytype σ: we allow impredicative instantiation. For example, consider the call $(map\ @(\forall a.a \to a)\ f)$, where we supply one of the two type arguments that *map* expects (its type is in Figure 1). The instantiation judgement will then look like:

$$\begin{array}{c}\Gamma \vdash^{\mathsf{inst}} (\forall p\ q.(p \to q) \to [p] \to [q])\ ;\ \ @(\forall a.a \to a), f \\ \rightsquigarrow\ ((\forall a.a \to a) \to \kappa)\ ;\ [\forall a.a \to a] \to [\kappa]\end{array}$$

Here we end up with just one instantiation variable κ, which instantiates q; the other quantifier p is directly instantiated by the supplied type argument.

The attentive reader may note that our typing rules are sloppy about the lexical scoping of type variables (for example in rule GEN), so that they can appear in user-written type signatures or type arguments. Doing this properly is not hard, using the approach of Eisenberg et al. [2018], but the plumbing is distracting so we omit it.

5 QUICK LOOK IMPREDICATIVITY

Building on the baseline of Section 4, we are now ready to present Quick Look, the main contribution of this paper. The changes to the typing rules are given in Figure 5. We make small changes, highlighted in grey, in two existing rules:

- *Learning from the arguments.* During instantiation, in rule IARG, we take a quick look at the argument, using a new judgement $\vdash_{\natural}$ to produce a poly-substitution Θ that expresses what Quick Look learned from the argument (Section 5.1).
- *Learning from the result.* In rule APP-$\Downarrow$ we match ρ_r, the result type of the call, with ρ, the type expected by the context, learning a poly-substitution Θ. The rule then exploits Θ but otherwise behaves exactly as before. We only learn from the result in *checking* mode (APP-$\Downarrow$), because only in checking mode do we have an expected result type to learn from.

Quick Look is focused exclusively on reducing the need for type annotations (in the form of VTA) when *instantiating a call of a polymorphic function*. For example, the rules for typing a lambda (ABS-$\Uparrow$ and ABS-$\Downarrow$) remain unaffected. Hence, just as before, in inference mode we can only infer a monotype for a lambda-bound variable.

[3] Note that we do not address all the details of the design of Eisenberg et al. [2016] In particular, we do not account for the difference between "specified"' and "inferred" quantifiers.

$$\boxed{\Gamma \vdash^{\mathsf{i}} \sigma \;;\; \overline{\pi} \rightsquigarrow \Theta \;;\; \overline{\phi} \;;\; \rho_r}$$

$$\frac{\Gamma \vdash_{\lightning} e : \sigma_1 \rightsquigarrow \Theta_1 \qquad \Gamma \vdash^{\mathsf{i}} \Theta_1\sigma_2 \;;\; \overline{\pi} \rightsquigarrow \Theta_2 \;;\; \overline{\phi} \;;\; \rho_r \qquad \Theta = \Theta_2 \circ \Theta_1}{\Gamma \vdash^{\mathsf{i}} (\sigma_1 \to \sigma_2) \;;\; e, \overline{\pi} \rightsquigarrow \Theta \;;\; \Theta\sigma_1, \overline{\phi} \;;\; \rho_r} \textsc{ iarg}$$

$$\boxed{\Gamma \vdash_{\Downarrow} e : \rho}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \rightsquigarrow \overline{\phi} \;;\; \rho_r \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \\ \Theta = \mathsf{mgu}_{\mathsf{ql}}(\rho_r, \rho) \qquad \overline{\phi}' = \overline{\Theta\phi} \qquad \mathsf{dom}(\theta) = \mathsf{fiv}(\overline{\phi}') \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi'_i}\end{array}}{\Gamma \vdash_{\Downarrow} h\,\overline{\pi} : \rho} \textsc{ app-}\Downarrow$$

$$\boxed{\text{QL argument} \quad \Gamma \vdash_{\lightning} e : \sigma \rightsquigarrow \Theta}$$

$$\frac{\Gamma \vdash^{\mathsf{h}}_{\lightning} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \rightsquigarrow \overline{\phi} \;;\; \rho_r \qquad \rho = \mathsf{T}\,\overline{\sigma} \text{ or } \mathsf{fiv}(\rho_r) = \emptyset}{\Gamma \vdash_{\lightning} h\,\overline{\pi} : \rho \rightsquigarrow \mathsf{mgu}_{\mathsf{ql}}(\rho, \rho_r)} \textsc{ app-}\lightning$$

$$\frac{\text{Rule } \textsc{app-}\lightning \text{ fails}}{\Gamma \vdash_{\lightning} e : \sigma \rightsquigarrow \emptyset} \textsc{ rest-}\lightning$$

$$\boxed{\text{QL head} \quad \Gamma \vdash^{\mathsf{h}}_{\lightning} h : \sigma}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}}_{\lightning} x : \sigma} \textsc{ h-var-}\lightning \qquad \frac{}{\Gamma \vdash^{\mathsf{h}}_{\lightning} (e :: \sigma) : \sigma} \textsc{ h-ann-}\lightning$$

Fig. 5. Quick Look rules

5.1 A Quick Look at the Argument

Rule IARG (Figure 5) uses $\vdash_{\lightning}$ to take a quick look at the argument e. This judgement has the form:

$$\Gamma \vdash_{\lightning} e : \phi \rightsquigarrow \Theta$$

Here e is the argument, ϕ is the type expected by the function (called σ_1 in IARG), and Θ is the knowledge (expressed as a poly-substitution for some of the instantiation variables) gained from comparing e with ϕ. The rules of this judgement are given in Figure 5.

Looking first at rule APP-$\lightning$, when the argument is itself a call ($h\,\overline{\pi}$) (including the degenerate case of a bare variable), and the expected type is a top-level-monomorphic type (the ρ in the conclusion), we perform Quick Look in four steps:

(1) Infer the type σ of the head h of the application. The judgement $\vdash^{\mathsf{h}}_{\lightning}$, also given in Figure 5, is extremely simple and somewhat similar to $\vdash^{\mathsf{h}}_{\Uparrow}$ in Figure 3.
(2) Instantiate the function type σ, given arguments $\overline{\pi}$, using the same $\vdash^{\mathsf{inst}}$ judgement as before.
(3) We will discuss the mysterious premise $\rho = \mathsf{T}\,\overline{\sigma}$ or $\mathsf{fiv}(\rho_r) = \emptyset$ in Sections 5.3 and 5.4. If neither holds we revert to REST-$\lightning$.

(4) Return the poly-substitution gained by unifying ρ_r, the result type of the call, with ρ, the type expected by the function, using a standard most-general-unifier $\mathsf{mgu}_{\mathsf{ql}}(\rho, \rho_r)$.

For example, when typechecking the call (*head ids*), we instantiate *head*'s type with κ, and take a quick look at the argument *ids*. For that, we look up $ids :: [\forall a.a \to a]$ in Γ, and unify *head*'s expected argument type $[\kappa]$ against it. That yields $\Theta = [\kappa := \forall a.a \to a]$.

Rule REST-$\lightning$ deals with the cases where APP-$\lightning$ does not apply. Specifically: if the expected type σ is not a ρ-type; if the argument is more complicated than a call (a list comprehension, case expression, lambda, etc – Section 3.2); or if the side conditions in APP-$\lightning$ fail. In effect, we simply give up on using that argument for Quick Look. Remember: the quick look produces a substitution if it is easy and unambiguous to do so; but it is always free to to bale out, producing an empty substitution. In the end APP-$\Downarrow$ will fill in any un-substituted instantiation variables with monotypes, via θ.

5.2 Quick Look Unification

The unification function $\mathsf{mgu}_{\mathsf{ql}}(\sigma, \phi)$ unifies two polytypes, returning a poly-substitution, binding instantiation variables (only) to polytypes. Ordinary type variables a are treated as skolems; they cannot be unified by $\mathsf{mgu}_{\mathsf{ql}}$.

Why do we need to *unify* during Quick Look, rather than just *one-way match*? Suppose

$$f :: \forall a.(\sigma, a) \to Int \qquad x :: \forall c.(c, c)$$

where σ is any polytype (e.g. $\forall b.b \to b$). Now consider the call ($f\ x$). In rule APP we instantiate f's type with $[a := \kappa]$. Now we take a quick look at the argument x in rule IARG. In turn we instantiate x's type (in rule APP-$\lightning$), with $[c := \mu]$. Finally we call $\mathsf{mgu}_{\mathsf{ql}}((\sigma, \kappa), (\mu, \mu))$. This is easily solved by the substitution $[\kappa := \sigma, \mu := \sigma]$, but note that we had to unify variables on *both sides of the call* to do so. The unification function needs to cope with polymorphic types. For example:

$$\begin{array}{l} \mathsf{mgu}_{\mathsf{ql}}(\forall a.\, a \to \kappa, \forall b.\, b \to Int) = [\kappa := Int] \\ \mathsf{mgu}_{\mathsf{ql}}(\forall a.\, a \to \kappa, \forall b.\, b \to b) \text{ fails} \end{array}$$

The second case fails, because κ cannot be bound to a locally-quantified variable. This kind of unification that checks for escaping of inner quantified variables is standard. For example Pottier and Rémy [2005] describe constraint solving with scope extrusion rules past universal quantifiers. A more general setting than ours, where unification variables can be introduced at arbitrary nested scopes, is mixed-prefix unification [Miller 1992].

5.3 Guarded Arguments

The purpose of rule APP-$\lightning$ is to discover if the argument $h\,\overline{\pi}$ of some enclosing call (corresponding to e from rule IARG) unambiguously specifies how to instantiate some variables of ρ (corresponding to σ_1 from IARG). It does so by unifying ρ, the type expected by the enclosing call, with ρ_r, the type of the argument $h\,\overline{\pi}$.

But this unification risks making an arbitrary choice among possible instantiations. Consider these two calls (the types are in Figure 1 as usual):

$$\text{Call 1: } (:)\ id\ ids \qquad\qquad \text{Call 2: } (:)\ id\ incs$$

In the first we must instantiate (:) with type $\forall a.a \to a$, while in the second we must instantiate it with $Int \to Int$. Plainly cannot discover this instantiation from the first argument, which is *id* in both cases. In contrast, the necessary instantiation becomes absolutely plain when we look at the second argument.

In Call 1, suppose we instantiate (:) with $[a := \kappa]$. Then the expected type of the first argument is κ, a bare variable, while the actual argument, *id*, has type $\forall a.a \to a$. So should we instantiate

κ with $\forall a.a \to a$? Or should we instantiate the argument *id*, and *then* bind its type to κ? Since there is more than one choice, we should make neither; perhaps one of the other arguments of the application will fix κ instead.

And so it proves: the expected type of the second argument is $[\kappa]$, while the actual argument, *ids*, has type $[\forall a.a \to a]$. In Call 2, the expected type is again $[\kappa]$, but the actual argument, *incs*, has type $Int \to Int$; so κ must be equal to $Int \to Int$. We say that κ appears *guarded* by the list type constructor, in the type of the second argument. That is the meaning of the premise $\rho = T\ \overline{\sigma}$ in APP-$\lightning$: if the argument type ρ is headed by a type constructor, we are free to unify.

5.4 Unguarded Arguments

Sometimes, though, the instantiation is unambiguous *even when the argument type is un-guarded.* Consider $single :: \forall a.a \to [a]$, and the call $(single\ \ ids)$. After instantiating with $[a := \kappa]$, *single* expects an argument of type κ, while the actual argument, *ids*, has type $[\forall a.a \to a]$. Despite the fact that *single*'s argument has an unguarded type, κ, there is no ambiguity here: the only possible instantiation is $[\kappa \mapsto [\forall a.a \to a]]$.

Why is this the only possible instantiation? Because the result type of the argument, $\rho_r = [\forall a.a \to a]$, has no free instantiation variables, so cannot be generalised, and hence no well-typed program can instantiate κ with a top-level forall. This is the role of the side condition $\mathsf{fiv}(\rho_r) = \emptyset$ in APP-$\lightning$ in Figure 5

Note that this condition works equally well when the argument is an application, not just a bare variable. For example consider: $g_1 :: \forall a.Bool \to a \to a$ and the application: $single\ (g_1\ True)$. In this case we'd get back $\rho_r = (\kappa_a \to \kappa_a)$, which has a free instantiation variable (κ_a), arising from instantiating g_1. We *should not* commit to a monomorphic $[\kappa := (\kappa_a \to \kappa_a)]$ because there's genuine ambiguity: another typing with $[\kappa := (\forall a.a \to a)]$ is also possible. The two choices would correspond to two different explicit System F derivations (after also fixing κ_a to, say, *Int*):

Instantiation 1	Instantiation 2
$single\ \ @(Int \to Int)\ (g_1\ \ @Int\ \ True)$	$single\ \ @(\forall a.a \to a)\ (\Lambda a.\ g_1\ \ @a\ \ True)$

Technically, the condition $\mathsf{fiv}(\rho_r) = \emptyset$ does not fully eliminate ambiguity due to implicit generalization, if *vacuous* quantification is allowed. Consider this expression:

$$single\ ids :: [\forall c.[\forall a.a \to a]]$$

Here APP-$\lightning$ will instantiate *single* at $[\forall a.a \to a]$. That contradicts the type signature, which has a vacuous quantification over c, so the program will be rejected in our system. We choose to ignore such vacuous possibilities, in exchange for making Quick Look a bit more powerful; for example, cases A3, A5, A7, and A12 in Figure 12 all require the side condition. This rather subtle point is a free design choice. We could simplify the system slightly by dropping the condition $\mathsf{fiv}(\rho_r) = \emptyset$ in APP-$\lightning$, but then these examples would require type annotations.

Here are some additional examples and the result of Quick Look:

	Call	where	QL argument instantiation
(1)	$single\ g_1$	$g_1 :: \forall a.Bool \to a \to a$	$\emptyset$
(2)	$single\ g_2$	$g_2 :: Bool \to \forall a.a \to a$	$[\kappa := Bool \to \forall a.a \to a]$
(3)	$single\ (g_2\ \ True)$		$\emptyset$
(4)	$single\ (g_3\ \ True)$	$g_3 :: \forall a.a \to (\forall b.b \to b) \to a$	$[\kappa := (\forall b.b \to b) \to Bool]$

To demonstrate that we should learn nothing from the argument of *single* in examples (1), and (3), here are two distinct valid instantiations for each, expressed in System F:

	Instantiation 1	Instantiation 2
(1)	$single\ @(Bool \to Int \to Int)\ (g_1\ @Int)$	$single\ @(\forall a.Bool \to a \to a)\ g_1$
(3)	$single\ @(Int \to Int)\ (g_2\ True\ @Int)$	$single\ @(\forall a.a \to a)\ (g_2\ True)$

Since there is more than one possibility, Quick Look should commit to neither, instead allowing some other argument (or result type) to unambiguously fix the instantiation. We return to this topic in Section Section 6.2.

5.5 Quick Look at the Result

So much for taking a quick look at arguments. As we saw in Section 3.1, we can also glean instantiation information from the result type, at least in checking mode. The highlighted part of rule APP-$\Downarrow$ in Figure 5 therefore uses $\mathsf{mgu}_{\mathsf{ql}}$ to match ρ_r, the result type of the call, with ρ, the result "pushed down" from the context by bidirectional type inference. We can use the same $\mathsf{mgu}_{\mathsf{ql}}$ function as before, but this time it really only performs matching, because ρ is devoid of instantiation variables.

For example, consider the expression $([] :: [\forall a.a \to a])$, where $[]$ is the empty list, whose type is $\forall p.[p]$ (Figure 1). The user-written type signature $[\forall a.a \to a]$ is pushed down by rule H-ANN (Figure 3). Then we find the (degenerate) call $[]$; instantiating its type with κ we get the result type $[\kappa]$. Rule APP-$\Downarrow$ now calls $\mathsf{mgu}_{\mathsf{ql}}([\kappa], [\forall a.a \to a])$, which succeeds with substitution $\Theta = [\kappa \mapsto \forall a.a \to a]$, as desired.

All this happens only in the checking case ($\delta = \Downarrow$). For the inference case, rule APP-$\Uparrow$ is unchanged, and we perform no Quick Look on the result type. We can't — we are *inferring* the result type. But even in inference mode, we will still take a quick look at the arguments, of course.

5.6 Over-saturated Functions

Consider the call (*head ids True*). This is a tricky one! We have $head :: \forall p.[p] \to p$, which looks as if it takes one argument, yet here it is applied to two. That is fine: *ids* is a list of polymorphic functions, so we extract the head, instantiate it, and apply the result to *True*. If we put in the type applications it would look like $(head\ \ @(\forall a.a \to a)\ \ ids\ \ @Bool\ \ True)$.

It is for this reason that rule IARG in Figure 5 applies the substitution Θ_1 (gleaned from Quick Look at the argument) to the result type σ_2 before instantiating the rest of the function. In this example, suppose we instantiate *head* with κ to yield the type $[\kappa] \to \kappa$. Then in rule IARG we take a quick look at the argument *ids*, yielding $\Theta_1 = [\kappa := \forall a.a \to a]$. We apply Θ_1 to the result type κ, giving $\forall a.a \to a$. Then we instantiate with ITYARG and use IARG on the argument *True*. In the end

$$\Gamma \vdash^{\mathsf{inst}} (\forall p.[p] \to p)\ ;\ ids, True \rightsquigarrow [\forall a.a \to a], Bool\ ;\ Bool$$

5.7 Argument Order

Suppose f_1 and f_2 are the same function, but take their two arguments in a different order. Then we would expect $(f_1\ e_1\ e_2)$ to typecheck if and only if $(f_2\ e_2\ e_1)$ typechecks; typechecking should not be sensitive to the vagaries of argument order. In the system we have presented so far this property does not quite hold. Consider:

$$\begin{array}{ll} f_1 :: \forall a.[a] \to a \to Int & x_1 :: \forall b.[[b]] \\ f_2 :: \forall a.a \to [a] \to Int & x_2 :: [\forall c.c \to c] \end{array}$$

Then $(f_1\ x_1\ x_2)$ typechecks, but $(f_2\ x_2\ x_1)$ does not. Why? Suppose we instantiate f_1 with $[a := \kappa]$, and its first argument x_1 with $[b := \mu]$. The first argument is guarded, so Quick Look learns

$[\kappa := [\mu]]$. Moving on to the second argument, we apply what we have learned so far (the $\Theta_1\sigma_2$ in IARG), so the expected type is not κ but $[\mu]$ *and that is guarded*. So we can learn $[\mu := \forall c.c \to c]$. But if we try to typecheck $(f_2\ x_2\ x_1)$, the first argument is *unguarded*, so we learn nothing. From the second we learn $[\kappa := [\mu]]$ as before, but that is all we get, and it is not enough to type the term. The difference arises because IARG applies the substitution Θ as it goes – and it *must* do so to handle over-saturated functions (Section 5.6).

There are several ways out of this dilemma. We could (1) *reject both calls*, by computing a guardedness flag for each argument based on the *uninstantiated* type of the function, and use those precomputed flags (unaffected by Quick Look's progress) to determine guardedness in rule APP-$\lightning$.

As a more complex alternative, we could (2) *accept both calls*, by typing guarded arguments first, and then unguarded ones (in the hope that they are now guarded). This is similar in spirit to the approach taken by HMF [Leijen 2008, §6.6]. Even more obscure examples show that one would have to iterate this process to a fixed point. Finally, we could (3) ignore the problem: the typing rules are simple and predictable, and the order-sensitive examples are pretty obscure.

In our implementation we use approach (1): it requires simpler reasoning from the programmer, and is straightforward to implement.

5.8 Co- and Contravariance of Function Types

The presentation so far treats the function arrow $(\to)$ uniformly with other type constructors T. Suppose that

$$f :: (\forall a.Int \to a \to a) \to Bool \qquad g :: Int \to \forall b.b \to b$$

Then the call $(f\ g)$ is ill-typed because we use equality when comparing the expected and actual result types in rule APP. The call would also be rejected if the foralls in f and g's types were the other way around. Only if they line up will the call be accepted. The function is neither covariant nor contravariant with respect to polymorphism; it is invariant.

We make this choice for three reasons. First, and most important for this paper, treating the function arrow invariantly means that it acts as a guard, which in turn allows more impredicative instantiations to be inferred. For example, without an invariant function arrow (*app runST argST*) cannot be typed.

Second, such mismatches are rare (we give data in Appendix A), and even when one occurs it can readily be fixed by η-expansion. For example, the call $(f\ (\lambda x.\ g\ x))$ is accepted regardless of the position of the foralls.

Finally, as well as losing guardedness, co/contra-variance in the function arrow imposes other significant costs. One approach, used by GHC, is to perform automatic η-expansion, through so-called "deep skolemisation" and "deep instantiation" [Peyton Jones et al. 2007, §4.6]. But, aside from adding significant complexity to the type system, this automatic η-expansion changes the semantics of the program (both in call-by-name and call-by-need settings), which is highly questionable.

Instead of *actually* η-expanding, one could make the type system behave *as if* η-expansion had taken place. This would, however, impact the compiler's intermediate language. GHC elaborates the source program to a statically-typed intermediate language based on System F; we would have to extend this along the lines of Mitchell's System Fη [Mitchell 1988], a major change that would in turn impact GHC's entire downstream optimisation pipeline.

In short, an invariant function arrow provides better impredicative inference, costs the programmer little, and makes the type system significantly simpler. Indeed, a GHC Proposal to simplify the language by adopting an invariant function arrow has been adopted by the community, independently of impredicativity [Peyton Jones 2019]. In Appendix A we quantify the impact of this change in the broader Haskell ecosystem.

5.9 Modularity

Quick Look is like many other works in that it exploits programmer-supplied type annotations to guide type inference (Section 11). But Quick Look's truly distinctive feature is that it is *modular* and *highly localised.*

Highly localised. Through half-closed eyes the changes in Figure 5 may seem substantial. However, rule APP-$\Downarrow$ is the only rule of the expression judgement that is changed. When scaling up to all of Haskell, the expression judgement in Figure 3 gains dozens and dozens of rules, one for each syntactic construct. But the only change to support Quick Look impredicativity remains rule APP-$\Downarrow$. So Quick Look scales well to a very rich source language.

Modular. This paper has presented only a minimalistic type system, but GHC offers many, many more features, including **let**-generalisation, data types and pattern matching, GADTs, existentials, type classes, type families, higher kinds, quantified constraints, kind polymorphism, dependent kinds, and so on. GHC's type inference engine works by generating constraints solving them separately, and elaborating the program into System F [Vytiniotis et al. 2011]. *All of these extensions, and the inference engine that supports them, are unaffected by Quick Look.* Indeed, we conjecture the Quick Look would be equally compatible with quite different type systems, such as ones involving subtyping, or dependent object types.

To substantiate these claims, Section 8 gives the extra rules for a much larger language; and we have built a full implementation in GHC (Section 9). This implementation is the first working implementation of impredicativity in GHC, despite several attempts over the last decade, each of which became mired in complexity.

6 PROPERTIES OF QUICK LOOK

In this section we give a comprehensive account of various properties of Quick Look.

6.1 Expressiveness and Backward Compatibility

Our system is able to type any program typeable in System F, maybe with additional annotations. In order to do so, we define a type-directed translation from System F into our source language, in a fashion very similar to Serrano et al. [2018]. Every variable, and application is recursively translated, and in addition:

- A System F abstraction $(\lambda(x :: \sigma).\, e)$ is translated as $(\lambda x.\, e' :: \sigma \to \phi)$, where e' is the translation of e, and ϕ is the type of e.
- A System F type application $(e\ @\sigma)$ is translated as $(e'\ @\sigma)$, where e' is the translation of e.
- A System F type abstraction $(\Lambda a.\ e)$ is translated as the annotated term $(e' :: \forall a.\sigma)$, where e' is the recursive translation of e, and σ is the type of e.

THEOREM 6.1 (EMBEDDING OF SYSTEM F). *Let e be a well-typed System F expression with type σ under an environment Γ, and e' the translation as defined above. Then $\Gamma \vdash_{\Downarrow} e' : \sigma$.*

The inverse translation, from our language into System F, is also simple to define. In particular, uses of the $\vdash^{\mathsf{inst}}$ judgment translate into type applications, and uses of $\vdash^{\forall}_{\Downarrow}$ translate into type abstractions. The fact that we elaborate to System F – a provably type-safe system where types can be erased – means that the proposed system is type-safe.

The theorem above shows that there exists a compositional translation of System F into typeable programs in Quick Look but it is heavy on type annotations. In terms of *practical guidance* about where type annotations may be needed, programmers need (1) annotate all lambdas with polymorphic argument types (or annotate just bound polymorphic variables if present in the syntax),

(2) give a visible type argument for every polymorphic (impredicative) instantiation that does not have a variable at the head of the instantiated function argument type, or that is not fixed by taking a quick look at the arguments or result type. This in turn means that although the naive translation of System F is heavy on type annotations, in practice very few annotations are needed.

Theorem 6.2 (Compatibility with rank-1 polymorphism). *Let e be an expression with type τ under λ-calculus with predicative polymorphism, in an environment Γ whose types only have top-level polymorphism. Then $\Gamma \vdash_{\Uparrow} e : \tau$ in the system presented in this paper.*

The theorem holds because, given the conditions on Γ, all the argument types $\overline{\phi}$ in rule APP will be monotypes, so Quick Look will recover no useful information, and will effectively be a no-op. Technically, Damas-Milner also includes generalizing **let** bindings, but adding those in our system poses no technical challenges (Appendix B).

6.2 Uniqueness of Quick Look

A crucial intuition is that if Quick Look returns a substitution of the instantiation variables then that is unambiguously the most general substitution we could use to type an application. For example, in Section 5.4 we showed examples where, since more than one instantiation is possible, we choose neither, thanks to the side-conditions in rule APP-$\lightning$.

This is of great practical importance, because if Quick Look for one argument makes an "incorrect" choice among multiple possibilities, that choice might contradict an unambiguous choice (the "correct" answer) gotten from another.

It would clearly be desirable to formalise this principle. Intuitively, the quick-look substitution should always be "on the way to" the substitution that would witness *any* valid typing derivation. But any valid typing derivation in what system? Presumably in a system allowing *arbitrary* polymorphic instantiation, as in System F. We have a theorem along these lines, but even the statement of the theorem requires significant technical scaffolding, including instrumented derivations in a variant of System F with n-ary applications, and a slightly different presentation of our instantiation judgement with an accumulating substitution, to name a few. Hence we leave a proper formal presentation of this result as future work.

6.3 Program Transformations

In this section we consider how several program transformations affect typeability in Quick Look.

Relating inference and checking mode. A desirable property of a type system is that if we can *infer* a type for a term then we can certainly *check* that the term can be assigned this type. The next theorem guarantees this; the proof is given in Appendix C.

Theorem 6.3. *If $\Gamma \vdash_{\Uparrow} e : \rho$ then $\Gamma \vdash_{\Downarrow} e : \rho$.*

As a consequence of this theorem, adding a type annotation (which changes the typechecking direction to checking) is a valid program transformation.

Let abstraction and inlining. One desirable property, at the heart of ML, is let-abstraction:

$$\textbf{let } x = e \textbf{ in } b \quad \equiv? \quad b\,[\,e \,/\, x\,]$$

This property does not hold in in our system; but nor does it hold in our baseline system PTIAT, because they use the context of the call to guide the typing of the argument. For example, suppose that $f :: ((\forall a.a \to a) \to Int) \to Bool$. Then let-abstracting the argument can render the program ill-typed. However, it can always be fixed by adding a type signature:

Unification variables		$\ni$	α, β, γ
Fully mono. types	τ	$::=$	$\alpha \mid \ldots$
Constraints	C	$::=$	$\epsilon \mid C \wedge C \mid \sigma \sim \phi \mid \forall \overline{a}.\, \exists \overline{\alpha}.\, C$

Fig. 6. Extra syntax for inference

$f\ (\lambda x.\ (x\ \mathit{True}, x\ 3))$	Well typed
$\mathbf{let}\ g = \lambda x.\ (x\ \mathit{True}, x\ 3)\ \mathbf{in}\ f\ g$	Not well typed
$\mathbf{let}\ g :: (\forall a. a \to a) \to \mathit{Int}$; $\quad g = \lambda x.\ (x\ \mathit{True}, x\ 3)$; $\mathbf{in}\ f\ g$	Well typed

What about the opposite of let-abstraction, namely let-inlining? With PTIAT, inlining a let-binding always improves typeability, but not so for Quick Look. Suppose $f :: \forall a. (\mathit{Int} \to a) \to a$. Then we have

$f\ (\lambda x.\ \mathit{ids})$	Not well typed
$\mathbf{let}\ g = \lambda x.\ \mathit{ids}\ \mathbf{in}\ f\ g$	Well typed
$f\ ((\lambda x.\ \mathit{ids}) :: \mathit{Int} \to [\forall a. a \to a])$	Well typed

The trouble here is that Quick Look does not look inside lambda arguments. Again, a type signature makes the program robust to such transformation, so that this equivalence always holds:

$$\mathbf{let}\ x = e :: \sigma\ \mathbf{in}\ b \quad \equiv \quad b\,[\,e :: \sigma\ /\ x\,]$$

η-expansion. Sometimes, as we have seen in Section 5.8, η-expansion is necessary to make a program typecheck. But sometimes the reverse is the case. For example, we cannot in general η-expand *runST* to become $(\lambda x.\ \mathit{runST}\ x)$, because then x would have to have a polytype, and in inference mode it can only have a monotype (rule APP-$\Uparrow$).

Argument permutation. As discussed in Section 5.7, Quick Look as described in Figure 5 is sensitive to permutation of function arguments. It is possible to achieve argument order independence by pre-computing a guardedness flag for each argument, and our implementation does so.

7 TYPE INFERENCE

In this section we give a *type inference algorithm* that implements the specification given in Section 5, and discuss its soundness.

7.1 Inference Algorithm

Typically type inference algorithms work in two stages: *generating* constraints, and then *solving* them, as described in *OutsideIn(X): Modular type inference with local assumptions* [Vytiniotis et al. 2011], which we abbreviate MTILA. To focus on impredicativity, we simplify MTILA by omitting local typing assumptions, along with data types, GADTs, and type classes – but our approach to impredicativity scales to handle all these features, as the full rules in Section 8 demonstrate.

Our algorithm generates *constraints* whose syntax is shown in Figure 6. Simple constraints are bags of *equality constraints* $\phi_1 \sim \phi_2$, but we will also need mixed-prefix constraints $\forall \overline{a}. \exists \overline{\alpha}. C$. These forms are not new; they are described in MTILA, and used in GHC's constraint solver. This is a key point of our approach: it requires zero changes to GHC's actual constraint language and solver.

$$\boxed{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C}$$

$$\frac{\Gamma, \overline{a} \vdash_{\Downarrow} e : \rho \rightsquigarrow C \qquad \overline{\alpha} = \mathsf{fuv}(C) - \mathsf{fuv}(\Gamma, \rho)}{\Gamma \vdash^{\forall}_{\Downarrow} e : \forall \overline{a}.\, \rho \rightsquigarrow \forall \overline{a}.\, \exists \overline{\alpha}.\, C}\ \textsc{gen}$$

$$\boxed{\Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow C}\ \boxed{\Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \rightsquigarrow C_{\mathsf{h}} \qquad \Gamma \vdash^{\mathsf{inst}} \sigma \,;\, \overline{\pi} \rightsquigarrow \overline{\phi} \,;\, \rho_r \,;\, C_{\mathsf{inst}} \\ \overline{\alpha}\ \text{fresh} \qquad \theta = [\mathsf{fiv}(\overline{\phi}, \rho_r) := \overline{\alpha}] \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi_i \rightsquigarrow C_i}\end{array}}{\Gamma \vdash_{\Uparrow} h\,\overline{\pi} : \theta\rho_r \rightsquigarrow C_{\mathsf{h}} \wedge C_{\mathsf{inst}} \wedge \overline{C_i}}\ \textsc{app-}\Uparrow$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \rightsquigarrow C_{\mathsf{h}} \qquad \Gamma \vdash^{\mathsf{inst}} \sigma \,;\, \overline{\pi} \rightsquigarrow \overline{\phi} \,;\, \rho_r \,;\, C_{\mathsf{inst}} \\ \Theta = \mathsf{mgu}_{\mathsf{ql}}(\rho_r, \rho) \qquad \overline{\phi}' = \overline{\Theta\phi} \qquad \rho_r' = \Theta\rho_r \\ \overline{\alpha}\ \text{fresh} \qquad \theta = [\mathsf{fiv}(\overline{\phi}', \rho_r') := \overline{\alpha}] \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi_i' \rightsquigarrow C_i}\end{array}}{\Gamma \vdash_{\Downarrow} h\,\overline{\pi} : \rho \rightsquigarrow C_{\mathsf{h}} \wedge C_{\mathsf{inst}} \wedge \overline{C_i} \wedge (\theta\rho_r' \sim \rho)}\ \textsc{app-}\Downarrow$$

$$\frac{\alpha\ \text{fresh} \qquad \Gamma, x : \alpha \vdash_{\Uparrow} e : \rho \rightsquigarrow C}{\Gamma \vdash_{\Uparrow} \lambda x.\, e : \alpha \to \rho \rightsquigarrow C}\ \textsc{abs-}\Uparrow$$

$$\frac{\beta_a, \beta_r\ \text{fresh} \qquad \Gamma, x : \beta_a \vdash_{\Downarrow} e : \beta_r \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.\, e : \alpha \rightsquigarrow C \wedge (\alpha \sim \beta_a \to \beta_r)}\ \textsc{absv-}\Downarrow \qquad \frac{\Gamma, x : \sigma_a \vdash^{\forall}_{\Downarrow} e : \sigma_r \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.\, e : \sigma_a \to \sigma_r \rightsquigarrow C}\ \textsc{absf-}\Downarrow$$

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} x : \sigma \rightsquigarrow \epsilon}\ \textsc{h-var} \qquad \frac{\Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow C}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \rho \rightsquigarrow C}\ \textsc{h-infer} \qquad \frac{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} (e :: \sigma) : \sigma \rightsquigarrow C}\ \textsc{h-annot}$$

Fig. 7. Inference algorithm: expressions

Algorithm: expressions. Figure 7 presents constraint generation for expressions. It closely follows the declarative specification in Figure 3, as modified in Figure 5. For example, the judgement $\Gamma \vdash_{\delta} e : \rho \rightsquigarrow C$ is very similar to that in Figure 3, but in addition generates constraints C. The big difference is that instead of clairvoyantly selecting monomorphic types τ for λ-abstraction arguments and for other instantiations (e.g. the range of substitution θ in rule APP-δ) the constraint-generation rules create fresh *unification variables*, α, β, γ. Unification variables stand for monomorphic types, and are solved during a subsequent *constraint solving* pass. In contrast, instantiation variables stand for polytypes, and are solved immediately by Quick Look; the constraint solver never sees them. Hence the following invariant:

LEMMA 7.1. *If* $\Gamma \vdash_{\delta} e : \rho \rightsquigarrow C$ *(with* $\mathsf{fiv}(\Gamma) = \emptyset$*) then* $\mathsf{fiv}(C) = \emptyset$.

Rule GEN generates a mixed-prefix constraint (a degenerate *implication constraint* in the MTILA jargon), that encodes the fact that the unification variables $\overline{\alpha}$ generated in C are allowed to unify to types mentioning the bound variables $\overline{a}$.

$$\boxed{\Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \leadsto \overline{\phi} \;;\; \rho_r \;;\; C}$$

$$\frac{\Gamma \vdash^{\mathsf{i}} \sigma \;;\; \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}{\Gamma \vdash^{\mathsf{inst}} \sigma \;;\; \overline{\pi} \leadsto \overline{\phi} \;;\; \rho_r \;;\; C}\ \textsc{inst}$$

$$\boxed{\begin{array}{c}\Gamma \vdash^{\mathsf{i}} \sigma \;;\; \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C \\ \text{Invariants: } \overline{\phi} \text{ and } \rho_r \text{ are fixed points of } \Theta\text{; } \mathsf{length}(\overline{\phi}) = \mathsf{length}(\overline{\pi})\end{array}}$$

$$\frac{}{\Gamma \vdash^{\mathsf{i}} \rho_r \;;\; \epsilon \leadsto \emptyset \;;\; \epsilon \;;\; \rho_r \;;\; \epsilon}\ \textsc{iresult}$$

$$\frac{\overline{\pi} \neq \sigma{,}\overline{\pi}' \quad \kappa \text{ fresh} \quad \Gamma \vdash^{\mathsf{i}} [a := \kappa]\rho \;;\; \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}{\Gamma \vdash^{\mathsf{i}} \forall a.\rho \;;\; \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}\ \textsc{iall}$$

$$\frac{\Gamma \vdash^{\mathsf{i}} \rho[a := \sigma] \;;\; \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}{\Gamma \vdash^{\mathsf{i}} \forall a.\rho \;;\; \sigma, \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}\ \textsc{ityarg}$$

$$\frac{\Gamma \vdash_{\lightning} e : \sigma_1 \leadsto \Theta_1 \qquad \Gamma \vdash^{\mathsf{i}} \Theta_1\sigma_2 \;;\; \overline{\pi} \leadsto \Theta_2 \;;\; \overline{\phi} \;;\; \rho_r \;;\; C \qquad \Theta = \Theta_2 \circ \Theta_1}{\Gamma \vdash^{\mathsf{i}} (\sigma_1 \to \sigma_2) \;;\; e, \overline{\pi} \leadsto \Theta \;;\; \Theta\sigma_1, \overline{\phi} \;;\; \rho_r \;;\; C}\ \textsc{iarg}$$

$$\frac{\beta, \gamma \text{ fresh} \quad \Gamma \vdash^{\mathsf{i}} (\beta \to \gamma) \;;\; e, \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}{\Gamma \vdash^{\mathsf{i}} \alpha \;;\; e, \overline{\pi} \leadsto \Theta \;;\; \overline{\phi} \;;\; \rho_r \;;\; C \wedge \alpha \sim (\beta \to \gamma)}\ \textsc{ivarm}$$

$$\frac{\mu, \upsilon \text{ fresh} \quad \Theta_1 = [\kappa := (\mu \to \upsilon)] \quad \Gamma \vdash^{\mathsf{i}} (\mu \to \upsilon) \;;\; e, \overline{\pi} \leadsto \Theta_2 \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}{\Gamma \vdash^{\mathsf{i}} \kappa \;;\; e, \overline{\pi} \leadsto \Theta_2 \circ \Theta_1 \;;\; \overline{\phi} \;;\; \rho_r \;;\; C}\ \textsc{ivard}$$

Fig. 8. Inference algorithm: instantiation

Rules ABSV-$\Downarrow$ and ABS-$\Uparrow$ generate fresh unification variables, as expected[4]. Note that they preserve the invariant that environments and constraints only mention unification variables but never instantiation variables.

Rules APP-$\Uparrow$ and APP-$\Downarrow$ follow their declarative counterparts in Figure 3 and Figure 5, with a few minor deviations. First, while rule APP-$\Downarrow$ in Figure 5 clairvoyantly selects a monomorphic θ to "monomorphise" any instantiation variables that are not given values by Quick Look, its algorithmic counterpart in Figure 7 generates fresh unification variables $\overline{\alpha}$. Second, rule APP-$\Downarrow$ generates further constraints about the result type; whereas the rule in Figure 5 has readily ensured that $\rho = \theta\rho_r$.

Algorithm: instantiation. The algorithmic instantiation judgement in Figure 8 collects constraints generated in rule IVARM. In that case we have a unification variable α that we have to further unify to a function type $\beta \to \gamma$. Note how the constraint C only mentions unification *but no instantiation variables.* Instantiation variables κ are born and eliminated in a single round of Quick Look.

[4] The alert reader will notice that ABSV-$\Downarrow$ is redundant; we can instead use APP-$\Downarrow$ with an empty $\overline{\pi}$ and $h = \lambda x.e$; that rule would in turn invoke $\vdash^{\mathsf{h}}_{\Uparrow} h : \sigma$ and thence (via H-INFER) land in ABS-$\Uparrow$. But this chain is pretty indirect, so we prefer to give ABSV-$\Downarrow$ directly.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \cdots \mid \text{case } e_0 \text{ of } \{\overline{K_i\,\overline{x_i} \to e_i}\} \\
\text{Constraints} & Q & ::= & \epsilon \mid Q \wedge Q \\
 & & \mid & \sigma \sim \phi \\
 & & \mid & \ldots \text{ extensible} \\
\text{Polymorphic types} & \sigma, \phi & ::= & \forall \overline{a}.\, Q \Rightarrow \rho \qquad \overline{a} \text{ and } Q \text{ may be empty} \\
\text{Constructor signatures} & \Bbbk & ::= & K : \forall \overline{a}\,\overline{b}.\, Q \Rightarrow \overline{\sigma} \to \mathsf{T}\,\overline{a} \\
\text{Environments} & \Gamma & ::= & \cdots \mid \Gamma, \Bbbk \mid \Gamma, Q
\end{array}
$$

Fig. 9. Syntax for extended language

7.2 Soundness of Type Inference

We write $\theta \models C$ to mean that a *monomorphic* idempotent substitution θ (from any sort of variables) is a solution to constraint C.[5] Due to Lemma 7.1, a solution θ to a constraint C need only refer to unification variables, but *never* instantiation variables, that must be resolved only through the QL mechanism. The main soundness theorem follows:

Theorem 7.2 (Soundness). *If* $\Gamma \vdash_\delta e : \rho \leadsto C$, θ *is a substitution from unification variables to monotypes,* $\mathsf{fiv}(\theta) = \emptyset$, *and* $\theta \models C$ *then* $\theta\Gamma \vdash_\delta e : \theta\rho$.

The theorem relies on a chain of other lemmas for every auxiliary judgement used in our specification and the algorithm. We give proofs in Appendix C.

We additionally conjecture that completeness is true of our algorithm, but have not attempted a detailed proof.

8 EXTENDING THE LANGUAGE WITH QUALIFIED TYPES AND GADTS

As discussed in Section 5.9, one of the salient features of Quick Look is its modularity with respect to other type system features. In this section we describe the integration with qualified types and GADTs, and leave **let** bindings for Appendix B.

Haskell has a much richer vocabulary of polymorphic types than simply $\forall \overline{a}.\, \rho$. In addition to quantified type variables, a set of *constraints* may appear, as described in Figure 9. Those constraints must be satisfied by the chosen instantiation in order for the program to be accepted.

Following Vytiniotis et al. [2011] we leave the language of constraints open; in the case of GHC this language includes type class constraints and equalities with type families. Figure 10 shows that the changes required to support qualified types are fairly minimal:

- Environments Γ may now also mention *local constraints*. This is a slight departure from Vytiniotis et al. [2011], in which variable environments and local constraints were kept in separate sets; this change allows us to control better the scope of each type variable.
- Rules GEN and IAPP now have to deal with constraints. In the former case, Q is added to the set of local constraints. In the case of IAPP, the constraints are returned as part of the instantiation judgment.
- Rule APP needs to check that the constraints obtained from instantiation hold for the chosen set of types. We use an auxiliary *constraint entailment* judgment $\Gamma \Vdash Q$ which states that constraints Q hold in the given environment (which may contain local assumptions). This judgment can be freely instantiated, as explained by Vytiniotis et al. [2011].

[5]For implication constraints θ is a substitution nesting – see [Serrano et al. 2018].

$$\boxed{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma}$$

$$\frac{\Gamma, \overline{a}, Q \vdash_{\Downarrow} e : \rho}{\Gamma \vdash^{\forall}_{\Downarrow} e : \forall \overline{a}.\ Q \Rightarrow \rho}\ \textsc{gen}$$

$$\boxed{\Gamma \vdash_{\Uparrow} e : \rho}\ \boxed{\Gamma \vdash_{\Downarrow} e : \rho}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma ; \overline{\pi} \rightsquigarrow Q ; \overline{\phi} ; \rho_r \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \\ \mathsf{dom}(\theta) = \mathsf{fiv}(\overline{\phi}, \rho_r) \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi_i} \qquad \rho = \theta\rho_r \qquad \Gamma \Vdash \theta Q\end{array}}{\Gamma \vdash_{\delta} h\,\overline{\pi} : \rho}\ \textsc{app-}\delta$$

$$\frac{\begin{array}{cc} & \text{for each branch } K_i\,\overline{x_i} \longrightarrow e_i \text{ do:} \\ \Gamma \vdash_{\Uparrow} e_0 : \mathsf{T}\,\overline{\sigma} & K_i : \forall \overline{a}\ \overline{b}.\, Q \Rightarrow \overline{v_i} \longrightarrow \mathsf{T}\,\overline{a} \in \Gamma \\ & \Gamma, \overline{x_i : [\overline{a := \sigma}]v_i}, \overline{b}, Q \vdash_{\delta} e_i : \rho \end{array}}{\Gamma \vdash_{\delta} \mathsf{case}\ e_0\ \mathsf{of}\ \{\overline{K_i\,\overline{x_i} \longrightarrow e_i}\} : \rho}\ \textsc{case}$$

$$\boxed{\Gamma \vdash^{\mathsf{i}} \sigma ; \overline{\pi} \rightsquigarrow \Theta ; Q ; \overline{\phi} ; \rho_r}$$

$$\frac{\overline{\pi} \neq \sigma,\overline{\pi}' \qquad \kappa \text{ fresh} \qquad \Gamma \vdash^{\mathsf{i}} ([a := \kappa]\rho) ; \overline{\pi} \rightsquigarrow \Theta ; Q' ; \overline{\phi} ; \rho_r}{\Gamma \vdash^{\mathsf{i}} (\forall a.Q \Rightarrow \rho) ; \overline{\pi} \rightsquigarrow \Theta ; \Theta Q \wedge Q' ; \overline{\phi} ; \rho_r}\ \textsc{iall}$$

$$\boxed{\text{Constraint entailment} \quad \Gamma \Vdash Q}$$

Fig. 10. Qualified types and GADTs

GADTs extend the language by allowing local constraints and quantification also in data type constructors. These constraints are in scope whenever pattern matching consider that case. The integration of pattern matching in a bidirectional type system can be done in several ways, depending on the direction in which the expression being matched is type checked. In the rule CASE in Figure 10 we look at that expression e_0 in inference mode; the converse choice is to look at how branches are using the value to infer the instantiation.

In principle, Quick Look is not affected by these changes. But we could also use some information about the usage of types in the rest of constraints to guide the choice of impredicativity. For example, Haskell does not allow type class instances over polymorphic types; so if we find a constraint *Eq a*, we know that *a* should not be impredicatively instantiated.

9 IMPLEMENTATION

One of our main claims is that Quick Look can be added, in a modular and non-invasive way, to an existing, production-scale type inference engine. To substantiate the claim, we implemented Quick Look on top of the latest incarnation of GHC (ghc-8.11.0.20200529). The changes were straightforward, and were highly localised. Overall we added about 450 lines to GHC's 90,000 line type inference engine. (These figures include comment-only lines, since they are a good proxy for tricky code.) By way of comparison, attempts to implement Guarded Instantiation [Serrano et al. 2018] in GHC floundered in a morass of complexity. The Quick Look implementation is publicly available.[6]

Our implementation completely avoids the potentially-quadratic cost of Quick Look (Section 3.2) by retaining and re-using the results of recursive quick looks. Happily, this turned out to require only simple and localised changes to our initial implementation of Quick Look.

Type classes proved to be the only really tricky point. Consider $wcs :: (HasCallStack \Rightarrow c) \rightarrow c$, and the innocent-looking call $(app\ wcs\ True)$. We instantiate app with $[a := \kappa, b := \mu]$. Then from app's first argument (which is guarded) we rightly conclude that $[\kappa := HasCallStack \Rightarrow \nu, \mu := \nu]$, where we instantiate wcs with ν. Quick Look is a no-op on the second argument because the expected argument type $HasCallStack \Rightarrow \nu$ is not a top-level monomorphic type, so APP-$\lightning$ rule does not apply. So the call typechecks fine.

But now consider the call $(revapp\ \ True\ \ wcs)$, where we reverse the argument order. After instantiating $revapp$ with $[a := \kappa, b := \mu]$, a quick look on the first argument succeeds, even though $revapp$'s first argument type is unguarded, because the type of the argument, $Bool$, has no free instantiation variables. So we conclude $[\kappa := Bool]$. Disaster: that instantiation does not allow the application to be typechecked. Permuting the arguments changes behaviour!

The problem is that the reasoning about unguarded arguments in Section 5.4 is undermined by type classes: although a vacuous *type* abstraction is not useful, a *type class* abstraction that binds no type variables may be very useful indeed. In this case, the quick look at the first argument of $revapp\ True\ wcs$ should not have yielded a substitution.

The simplest way to restore insensitivity to argument order is to drop the condition $\mathsf{fiv}(\rho_r)$ from APP-$\lightning$ entirely, along with Section 5.4. We are reluctant to do this, because it means that some apparently-simple examples (including ones in existing libraries) are no longer typeable without annotations. Instead, we accept a measure of order-dependence in applications, by recognising that the first argument of app has already fixed a to be a qualified type, and so APP-$\lightning$ should not apply to the second. But we stress that this is a free (and debatable) design choice.

10 APPLICATIONS

A reasonable question to ask is how the implementation of Quick Look in GHC discussed in Section 9 benefits the user. It is hard to say how useful a feature will be in practice when it does not yet exist, but we can give some indicative data.

First, GHC has had an unreliable, unsupported, and entirely un-specified implementation of impredicativity for many years. We scanned the source code of a collection of packages obtained from Stackage (LTS 13.6). This repository contains 607 packages that that use the extensions *RankNTypes*, *Rank2Types* or *ImpredicativeTypes*. Of these we found 20 that used *ImpredicativeTypes*, suggesting that impredicativity is regarded as nigh essential by some authors. Of these 20 packages, 10 compiled directly with our new GHC, 4 needed eta expansions due to GHC now using an invariant arrow (see Appendix A for more details), and 6 could not be compiled due to missing dependencies and other issues unrelated to our work.

Second, in our set of 607 packages, we identified 48 packages with at least one source file that had **newtype** and **forall** on the same line, indicating that the programmer had used a wrapper for a higher-rank type. Our scan was rather superficial, and we will have missed a number of cases, e.g., if a line contains **newtype** and a synonym that hides the **forall**.

[6]https://gitlab.haskell.org/ghc/ghc/-/merge_requests/3220

The questions then is: can we get rid of such wrapper types? We investigated in some detail 26 out of those 48 packages[7]. For 17 out of 26 packages, all wrapped types were used in an instance declaration. Quick Look has nothing to say here, making this is an important future direction.

In 6 of the remaining 9 cases, removing the wrapping on higher-rank types was an unqualified success. Consider the well-known Scrap Your Boilerplate package *syb*, that is depended on by more than 300 packages on Hackage.[8] In module *Data.Generics.Aliases* we find several types that wrap a higher-rank type:

$$\begin{array}{ll}
\textbf{type}\ \mathit{GenericM}\ m = \forall a.\mathit{Data}\ a \Rightarrow a \rightarrow m\ a & \textbf{newtype}\ \mathit{GenericM}'\ m = \mathit{GM}\ \{\mathit{unGM} :: \mathit{GenericM}\ m\} \\
\textbf{type}\ \mathit{GenericQ}\ r\ = \forall a.\mathit{Data}\ a \Rightarrow a \rightarrow r & \textbf{newtype}\ \mathit{GenericQ}'\ r\ = \mathit{GQ}\ \{\mathit{unGQ} :: \mathit{GenericQ}\ r\} \\
\textbf{type}\ \mathit{GenericT}\ \ = \forall a.\mathit{Data}\ a \Rightarrow a \rightarrow a & \textbf{newtype}\ \mathit{GenericT}'\ \ = \mathit{GT}\ \{\mathit{unGT} :: \mathit{GenericT}\}
\end{array}$$

These **newtype**s are exported and (only) used in the module *Data.Generics.Twins*. Here is a use of *GenericM′*, seen in the use of *GM* and *unGM* (we have elided some irrelevant code):

$$\begin{array}{l}
\mathit{gzipWithM} :: \forall m.\mathit{Monad}\ m \Rightarrow \mathit{GenericQ}\ (\mathit{GenericM}\ m) \rightarrow \mathit{GenericQ}\ (\mathit{GenericM}\ m) \\
\mathit{gzipWithM}\ f\ x\ y = \textbf{case}\ \mathit{gmapAccumM}\ \mathit{perkid}\ \mathit{funs}\ y\ \textbf{of}\ \ldots \\
\quad \textbf{where}\ \mathit{perkid}\ a\ d = (\mathit{tail}\ a, \mathit{unGM}\ (\mathit{head}\ a)\ d) \\
\qquad\qquad \mathit{funs} = \mathit{gmapQ}\ (\lambda k \rightarrow \mathit{GM}\ (f\ k))\ x
\end{array}$$

Our goal is to get rid of *GenericM′* and replace it with *GenericM*, in other words, to remove the wrapping *GM* and unwrapping *unGM* from the definitions of *perkid* and *funs*. To do so we have to add signatures, since polymorphic types are never inferred for function arguments, thus:

$$\begin{array}{l}
\textbf{where}\ \mathit{perkid} :: \forall b.\mathit{Data}\ b \Rightarrow [\mathit{GenericM}\ m] \rightarrow b \rightarrow ([\mathit{GenericM}\ m], m\ b) \\
\qquad\quad \mathit{perkid}\ a\ d = (\mathit{tail}\ a, (\mathit{head}\ a)\ d) \\
\qquad\quad \mathit{funs} :: [\mathit{GenericM}\ m] \\
\qquad\quad \mathit{funs} = \mathit{gmapQ}\ f\ x
\end{array}$$

These type signatures are desirable anyway, making the code far more comprehensible. We did the same for the other two types, *GenericQ′* and *GenericT′*, and were able to *completely remove all three auxiliary newtypes from the implementation.*

Three of the remaining packages were harder to deal with. In the case of *streamly* one wrapper type was easily removed, but two other wrapped types were harder to deal with because of the interaction between Quick Look and type families, and the desugaring of **do** notation. For *fixed-vector*, one type had a class instance problem, one type could be dealt with, essentially by replacing pattern matches by function calls to help Quick Look along. Interestingly, this package depended on the *primitive* package we had already successfully dealt with. The changes to the latter did not lead to issues with *fixed-vector*. For *reflection* the results are still inconclusive.

11 RELATED WORK

This section explores many different strands of work on first-class polymorphism; another excellent review can be found in [Botlan and Rémy 2009, §5]. Figure 11 shows how various impredicative systems impinge on users, by requiring them to understand new term forms, new type forms, or new type constraints. The fourth column shows whether type annotations may be necessary, beyond the binders of polymorphic lambdas. In this figure "no" is good! QL does not require any new terms, types, or constraints; but in exchange, QL sometimes requires an annotation on a non-guarded instantiation. HMF achieves a similar combination of features, but QL is better suited

[7]Of the remaining 22, 18 depended on packages we could not compile, because of different reasons, and in 4 cases the wrapped type was in code that was not used in a standard compile.
[8]https://packdeps.haskellers.com/reverse/syb version 0.7.1.

Name	New terms	New types	New constraints	Annotation needed beyond poly. lambdas
QL (this paper)	No	No	No	Yes
GI [Serrano et al. 2018]	No	No	Yes	Yes
MLF [Botlan and Rémy 2003]	No	Yes	Yes	No
HMF [Leijen 2008]	No	No	No	Yes
FPH [Vytiniotis et al. 2008]	No	No	Yes	No
HML [Leijen 2009]	No	Yes	Yes	No
Boxy [Vytiniotis et al. 2006]	No	Yes	No	Yes
QML [Russo and Vytiniotis 2009]	No	Yes	No	Yes
FreezeML [Emrich et al. 2019]	Yes	No	No	No
PolyML [Garrigue and Rémy 1999]	Yes	No	No	No

Fig. 11. Various impredicative systems features ("No" is good)

for the type inference engine of GHC since the quick look is a separate step before actual constraint generation. We continue below with a detailed discussion of related work for polymorphic type inference.

Figure 12 compares the expressiveness of some of these systems, using examples from their papers and more. As we see, QL performs well despite its simplicity.

Higher-rank polymorphism. Type inference for higher-rank polymorphism (in which foralls can appear to the left or right of the function arrow) is a well-studied topic with successful solutions using bidirectional type inference as discussed in Section 4 [Peyton Jones et al. 2007]. Follow-up modern presentations [Dunfield and Krishnaswami 2013] re-frame the problem within a more logical setting, and describe extensions to indexed types [Dunfield and Krishnaswami 2019]. The "C" examples are all about higher-rank inference, and mostly do not use impredicativity at all.

Boxed polymorphism. Impredicativity goes beyond higher-rank, by allowing quantified types to instantiate polymorphic types and data structures. Both Haskell and OCaml have supported impredicative polymorphism, in an inconvenient form, for over a decade.

In Haskell, one can wrap a polytype in a new, named data type or newtype, which then behaves like a monotype [Odersky and Läufer 1996]. This *boxed polymorphism* mechanism is easy to implement, but the programmer has to declare new data types and explicitly box and unbox the polymorphic value. Nevertheless, boxed polymorphism is widely used in Haskell.

OCaml supports *polymorphic object methods*, based on the theory of Poly-ML [Garrigue and Rémy 1999]. The programmer does not have to declare new data types, but polymorphic values must still be wrapped and unwrapped. In practice, the mechanism is little used, perhaps because it is only exposed through the object system.

In FreezeML [Emrich et al. 2019] the programmer chooses explicitly when to not instantiate a polymorphic type by using the freeze operator $\lceil - \rceil$, similar to how it is done in FX-89 [OâĂŹToole and Gifford 1989] with explicit `open` and `close` operations. Another variant of this line of work is QML [Russo and Vytiniotis 2009], which has two different universal quantifiers, one that can be implicitly instantiated and one that requires explicit instantiation. Introducing and eliminating the explicit quantifier is akin to the wrapping and unwrapping. Explicit wrapping and unwrapping may be tedious, especially since it often seems unnecessary, so our goal is to allow polymorphic functions to be implicitly instantiated with quantified types.

		QL	GI	MLF	HMF	FPH	HML
A	POLYMORPHIC INSTANTIATION						
A1	$const2 = \lambda x\ y.\ y$	✓	✓	✓	✓	✓	✓
	MLF infers $(b \geqslant \forall c.\ c \to c) \Rightarrow a \to b$, QL and GI infer $a \to b \to b$.						
A2	$choose\ id$	✓	✓	✓	✓	✓	✓
	MLF and HML infer $(a \geqslant \forall b.\ b \to b) \Rightarrow a \to a$, FPH, HMF, QL, and GI infer $(a \to a) \to a \to a$.						
A3	$choose\ [\,]\ ids$	✓	✓	✓	✓	✓	✓
A4	$\lambda(x :: \forall a.\ a \to a).\ x\ x$	✓	✓	✓	✓	✓	✓
	MLF infers $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$, QL and GI infer $(\forall a.\ a \to a) \to b \to b$.						
A5	$id\ auto$	✓	✓	✓	✓	✓	✓
A6	$id\ auto'$	No	✓	✓	✓	✓	✓
A7	$choose\ id\ auto$	✓	✓	✓	✓	No	✓
A8	$choose\ id\ auto'$	No	No	✓	No	No	✓
	QL and GI need an ann. on $id :: (\forall a.\ a \to a) \to (\forall a.\ a \to a)$.						
A9	$f\ (choose\ id)\ ids$	✓	✓*	✓	No	✓	✓
	where $f :: \forall a.\ (a \to a) \to [a] \to a$						
	GI needs an annotation on $(choose\ id) :: (\forall a.\ a \to a) \to (\forall a.\ a \to a)$.						
A10	$poly\ id;\ \ poly\ (\lambda x.\ x);\ \ id\ poly\ (\lambda x.\ x)$	✓	✓	✓	✓	✓	✓
A11	$k\ (\lambda f.\ (f\ 1, f\ True)\ xs$	✓	No	No	No	No	No
	where $k :: \forall a.\ a \to [a] \to Int$, $xs :: [(\forall a.\ a \to a) \to (Int, Bool)]$						
	Note the example requires impredicativity **and** bidirectionality.						
A12	$poly\ id;\ \ app\ poly\ id;\ \ revapp\ id\ poly$	✓	✓	✓	✓	✓	✓
A13	$app\ runST\ argST;\ \ revapp\ argST\ runST$	✓	✓	✓	✓	✓	✓
B	FUNCTIONS ON POLYMORPHIC LISTS						
B1	$length\ ids;\ \ tail\ ids;\ \ head\ ids;\ \ single\ id$	✓	✓	✓	✓	✓	✓
B2	$id : ids$	✓	✓	✓	✓†	✓	✓
B3	$(\lambda x.\ x) : ids$	✓	✓	✓	✓†	✓	✓
B4	$single\ inc +\!\!+ single\ id$	✓	✓	✓	✓	✓	✓
B5	$single\ id +\!\!+ ids$	✓	No	✓	No	✓	✓
B6	$map\ poly\ (single\ id)$	✓	No	✓	✓	✓	✓
	GI needs an ann. on $single\ id :: [\forall a.\ a \to a]$ in the previous two.						
B7	$map\ head\ (single\ ids);\ \ head\ ids\ True$	✓	✓	✓	✓	✓	✓
C	INFERENCE OF POLYMORPHIC LAMBDA BINDERS AND GENERALIZATION POINTS						
C1a	$\lambda f.\ (f\ 1, f\ True)$	No	No	No	No	No	No
C1b	$\lambda(f :: \forall a.\ a \to a).\ (f\ 1, f\ True)$	✓	✓	✓	✓	✓	✓
C1c	$g\ (\lambda f.\ (f\ 1, f\ True))$	✓	No	No	✓†	No	No
	where $g :: ((\forall a.\ a \to a) \to (Int, Bool)) \to Char$						
C2	$r\ (\lambda x\ y.\ y)$	✓	✓*	✓	✓†	No	No
	where $r :: (\forall a.\ a \to \forall b.\ b \to b) \to Int$						
E	η-EXPANSION $\quad k :: \forall a.\ a \to [a] \to a,\ \ h :: Int \to \forall a.\ a \to a,\ \ lst :: [\forall a.\ Int \to a \to a]$						
E1a	$k\ h\ lst$	No	No	No	No	No	No
E1b	$k\ (\lambda x.\ h\ x)\ lst$	✓	✓	✓	No	✓	✓
E2a	$\lambda x.\ poly\ x$	No	No	✓	No	No	No
E2b	$(\lambda x.\ poly\ x) :: (\forall a.\ a \to a) \to (Int, Bool)$	✓	No	✓	✓†	No	No
E3a	$app\ poly\ id$	✓	✓	✓	✓	✓	✓
E3b	$app\ (\lambda x.\ poly\ x)\ id$	No	No	✓	No	No	No
E4a	$map\ poly\ ids$	✓	✓	✓	✓	✓	✓
E4b	$map\ (\lambda x.\ poly\ x)\ ids$	✓	No	✓	No	No	No
E5a	$compose\ poly\ head$	✓	✓	✓	✓	✓	✓
E5b	$\lambda xs.\ poly\ (head\ xs)$	No	No	✓	No	No	No

* in GI requires extensions [Serrano et al. 2018]. † in HMF requires n-ary apps. and annotation propagation [Leijen 2008].

Fig. 12. Comparison of impredicative type systems

Guarded impredicativity. Quick Look builds on ideas originating in previous work on Guarded Impredicative Polymorphism (GI) [Serrano et al. 2018]. Specifically, GI figures out the polymorphic instantiation of variables that are "guarded" in the type of the instantiated function, or the types of the arguments; meaning they occur under some type constructor. However, GI relied on extending the constraint language with two completely new forms of constraints, one of which (delayed generalisation) was very tricky to conceptualise and implement. With Quick Look we instead eagerly figure out impredicative instantiations, quite separate from constraint solving, which can remain unmodified.

HMF [Leijen 2008] comes the closest to Quick Look in terms of expressiveness, with a simple type vocabulary, and an equation-based unification algorithm. HMF presents a declarative application typing rule that requires the "polymorphic weight" of the instantiated function type to be minimal, a condition encoding the fact that polymorphism cannot be guessed. The paper includes an extension of the basic system with n-ary applications, performing a subsumption between each function argument type and the (inferred and eagerly generalized) type of the corresponding argument. The order in which to perform these subsumption checks is delicate: it is first performed for guarded arguments and then on naked ones – this helps to establish some order-independence properties. All this bears a strong similarity to QL and to GI, but there is a significant difference that affects expressiveness: in Quick Look we more aggressively combine the propagation of annotations and impredicative instantiation with a deep quick look into *nested* applications before we perform traditional type inference. Hence we can type programs that require nested impredicative instantiation (such as B5 in Table 12) and programs that must be typed with a polymorphic binder in the environment but whose type we can only deduce via some other impredicative instantiation (such as E4b). There are other implementation differences; for example Quick Look does not need to generalize eagerly every argument in an application.

Stratified inference. The idea of Quick Look as a restricted pass prior to actual type inference has appeared before in the work on Stratified Type Inference (STI) [Pottier and Régis-Gianas 2006; Rémy 2005]. In the first pass, each term is annotated with a *shape*, a form of type that expresses the *quantifier structure* of the term's eventual type, while leaving its monomorphic components as flexible unification variables that will be filled in by the (conventional, predicative) second pass. To avoid shortcomings with the order in which arguments are checked this process may need to be iterated [Pottier and Régis-Gianas 2006, §7]. STI was probably the first work that demonstrated how annotation propagation in the form of a shape inference pass could be used to recover some impredicativity [Rémy 2005], using an order-dependent resolution of impredicative instantiations. QL has a similar flavor, but instead of performing annotation propagation *before* inference, it interweaves the two phases. It remains an interesting future direction to describe formally our system as an interweaving of these two separate mechanisms.

Beyond System F. Move beyond System F types impacts the language the programmer sees, the type inference algorithm, and the compiler's statically-typed intermediate language. However, once that Rubicon is crossed, there is a rich seam of work in systems with more expressive types or more expressive unification algorithms than first-order unification. The gold standard is MLF [Botlan and Rémy 2003], but there are several subsequent variants, including HML [Leijen 2009], and FPH [Vytiniotis et al. 2008].

MLF extends type schemes with instantiation constraints, and makes the unification algorithm aware of them. As a result it achieves the remarkable combination of: (i) typeability of the whole of System F by only annotating function arguments that must be used polymorphically, (ii) principal types and a sound and complete type inference algorithm, (iii) the "defining" ML property that any sub-term can be lifted and **let**-bound with no type annotations, without affecting typeability. In terms of expressiveness, MLF will *unconditionally* accept programs with polymorphic binders in the environment without any annotations if these arguments are not really used polymorphically (example E3b in Figure 12). This subsumes uses of η-expansion for functions with polymorphic argument types (example E2a) that will fail to be inferred in QL. On the other hand because QL uses the impredicative instantiation to *check* the arguments (*via* the bidirectional mechanism of Section 4) it can type some programs that involve polymorphic binders that are genuinely used at multiple types (example A11). It is worth mentioning that it is entirely possible to extend MLF with bidirectional propagation of annotations or even inferred polymorphic types (which would also type examples like A11 and C1c); in fact some small amount of annotation propagation has been implemented in MLF prototypes. However, such extensions would cause MLF to lose its simple specification and easy to describe typeability guarantees.

MLF and variants also require intrusive modifications to a constraint solver (in the case of GHC, a complex one with type classes, implication constraints, type families, and more) and to the type structure. Though some attempts have been made to integrate MLF with qualified types [Leijen and Löh 2005], a full integration is uncharted territory. Quick Look is a pragmatic compromise, trading off a little expressiveness for a lot of simplicity and ease of integration in the existing GHC inference engine.

ACKNOWLEDGMENTS

We warmly thank Richard Eisenberg, Matthew Fluet, Sam Lindley, Stephanie Weirich, Edward Yang, and the ICFP reviewers for their feedback. We are deeply grateful to Didier Rémy for his particularly detailed review and subsequent email dialogue, going far beyond the call of duty.

REFERENCES

Lennart Augustsson. 2011. Impredicative polymorphism: a use case. http://augustss.blogspot.com/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. https://doi.org/10.1145/944705.944709

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

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

Joshua Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290322

Richard A. Eisenberg, Joachim Breitner, and Simon Peyton Jones. 2018. Type variables in patterns. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018.* 94–105. https://doi.org/10.1145/3242744.3242753

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

Frank Emrich, Sam Lindley, Jan Stolarek, and James Cheney. 2019. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. Presented at TyDe 2019.

Jacques Garrigue and Didier Rémy. 1999. Semi-Explicit Higher-Order Polymorphism for ML. *Information and Computation* 155, 1/2 (1999), 134–169. http://www.springerlink.com/content/m303472288241339/ A preliminary version appeared in TACS'97.

Jurriaan Hage and Bastiaan Heeren. 2009. Strategies for Solving Constraints in Type and Effect Systems. *Electronic Notes in Theoretical Computer Science* 236 (2009), 163 – 183. Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures (VODCA 2008).

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008.* 283–294.

https://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*. 66–77. https://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. https://doi.org/10.1145/1086365.1086385

Dale Miller. 1992. Unification under a Mixed Prefix. *J. Symb. Comput.* 14, 4 (Oct. 1992), 321âĂŞ358. https://doi.org/10.1016/0747-7171(92)90011-R

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

John C. Mitchell. 1988. Polymorphic Type Inference and Containment. *Inf. Comput.* 76, 2-3 (Feb. 1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

Martin Odersky and Konstantin Läufer. 1996. Putting type annotations to work. In *Principles of Programming Languages, POPL*. 54–67.

J. W. OâĂŹToole 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* (Portland, Oregon, USA) *(PLDI âĂŹ89)*. Association for Computing Machinery, New York, NY, USA, 207âĂŞ217. https://doi.org/10.1145/73141.74836

Simon Peyton Jones. 2019. GHC Proposal: "Simplify subsumption". https://github.com/ghc-proposals/ghc-proposals/pull/287

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. 1995. On the Undecidability of Partial Polymorphic Type Reconstruction. *Fundamenta Informaticae* 19 (1995). Issue 1/2.

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

François Pottier and Yann Régis-Gianas. 2006. Stratified Type Inference for Generalized Algebraic Data Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. ACM, New York, NY, USA, 232–244. https://doi.org/10.1145/1111037.1111058

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

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

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proc ACM SIGPLAN Conference on Programming Languages Design and Implementation*. ACM. https://www.microsoft.com/en-us/research/publication/guarded-impredicative-polymorphism/

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

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

Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 295–306. https://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.

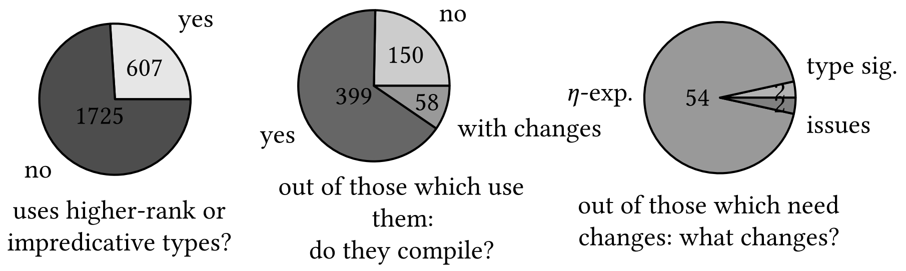

Fig. 13. Summary of the evaluation, using Stackage LTS 13.6

A IMPACT OF AN INVARIANT FUNCTION ARROW

In Section 5.8 we propose to make the function arrow invariant, and to drop deep instantiation and deep skolemisation. This change is independently attractive, and is the subject of a recently adopted GHC Proposal. How much effect does this change have on existing Haskell code?

To answer this question we used our implementation to compile a large collection of packages from Hackage; we summarize the results in Figure 13. We started from collection of packages for a recent version of GHC, obtained from Stackage (LTS 13.6), containing a total of 2,332 packages. We then selected only the 607 packages that used the extensions *RankNTypes*, *Rank2Types* or *ImpredicativeTypes*; the other 1,725 packages are certainly unaffected. We then compiled the library sections of each package. Of the 607 packages, 150 fail to compile for reasons unrelated to Quick Look – they depend on external libraries and tools we do not have available; or they do not compile with GHC 8.9 anyway. Of the remaining 457 packages, 399 compiled with no changes whatsoever; two had issues we could not solve, e.g., because of Template Haskell. We leave these two to the authors of these packages.

The remaining 56 packages could be made compilable with modest source code changes, almost all of which were a simple η-expansion on a line that was clearly identified by the error message. In total we performed 283 η-expansions in 104 of the total of 963 source files. The top three packages in this case needed 7, 7 and 9 files changed. The majority of the packages, 37, needed only one file to be changed, and 20 packages needed only a single η-expansion. In the case of two packages, *massiv* and *drinkery*, we additionally had to provide type signatures for local definitions.

In conclusion, of the 2,332 packages we started with, 74% (1,725/2,332) do not use extensions that interact with first-class polymorphism; of those that do, 87% (399/457) needed no source code changes; of those that needed changes, 97% (56/58) could be made to compile without any intimate knowledge of these packages. All but two were fixed by a handful of well-diagnosed η-expansions, two of them also needed some local type signatures.

B LOCAL BINDINGS

Our description of local bindings, whose syntax is given in Figure 14, follows Vytiniotis et al. [2011] closely. As described in Figure 15a, the type of a **let** binding is not generalized unless an explicit annotation is given. This design enables the most information to propagate between the definition of a local binding and its use sites when using a constraint-based formulation.

Figure 15b shows another possible design, in which generalization is performed on non-annotated **let**s. The main disadvantage is that when implemented in a constraint-based system, this forces us to solve the constraints obtained from e_1 before looking at e_2. Otherwise, we cannot be sure about

$$\begin{array}{llll} \text{Expressions} & e & ::= & \ldots \\ & & | & \mathsf{let}\ x = e\ \mathsf{in}\ e \\ & & | & \mathsf{let}\ x :: \sigma = e\ \mathsf{in}\ e \end{array}$$

Fig. 14. Syntax for local bindings

$$\boxed{\Gamma \vdash_{\Uparrow} e : \rho} \boxed{\Gamma \vdash_{\Downarrow} e : \rho}$$

$$\frac{\Gamma \vdash_{\Uparrow} e_1 : \rho_1 \qquad \Gamma, x : \rho_1 \vdash_{\delta} e_2 : \rho_2}{\Gamma \vdash_{\delta} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \rho_2}\ \textsc{let}$$

$$\frac{\Gamma \vdash^{\forall}_{\Downarrow} e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash_{\delta} e_2 : \rho_2}{\Gamma \vdash_{\delta} \mathsf{let}\ x :: \sigma_1 = e_1\ \mathsf{in}\ e_2 : \rho_2}\ \textsc{annlet}$$

(a) No generalization

$$\frac{\Gamma \vdash_{\Uparrow} e_1 : \rho_1 \qquad \overline{a} = \mathsf{fv}(\rho_1) - \mathsf{fv}(\Gamma) \qquad \Gamma, x : \forall \overline{a}.\, \rho_1 \vdash_{\delta} e_2 : \rho_2}{\Gamma \vdash_{\delta} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \rho_2}\ \textsc{letgen}$$

(b) With generalization

Fig. 15. Local bindings

which type variables to generalize. Another possibility, taken by Pottier and Rémy [2005] and Hage and Heeren [2009], is to extend the language of constraints with generalization and instantiation, making the solver aware of the order in which these constraints ought to be solved.

C PROOFS

C.1 Relating Inference and Checking Mode (Section 6.3)

Theorem C.1. *If* $\Gamma \vdash_{\Uparrow} e : \rho$ *then* $\Gamma \vdash_{\Downarrow} e : \rho$.

Proof. The proof is straightforward induction on the typing derivation. The interesting case is the case for rule app-$\Uparrow$. In that case we have:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\phi}\ ;\ \rho_r \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \\ \mathsf{dom}(\theta_1) = \mathsf{fiv}(\overline{\phi}, \rho_r) \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta_1 \phi_i}\end{array}}{\Gamma \vdash_{\Uparrow} h\, \overline{\pi} : \theta_1 \rho_r}\ \textsc{app-}\Uparrow$$

We need to show that $\Gamma \vdash_{\Downarrow} h\,\overline{\pi} : \theta_1 \rho_r$ by using rule app-$\Downarrow$. To do that it suffices to form the following derivation:

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \Gamma \vdash^{\mathsf{inst}} \sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\phi}\ ;\ \rho_r \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \\ \Theta = \mathsf{mgu}_{\mathsf{ql}}(\rho_r, \theta_1 \rho_r) \qquad \mathsf{dom}(\theta_2) = \mathsf{fiv}(\overline{\Theta \phi}) \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta_2 \Theta \phi_i}\end{array}}{\Gamma \vdash_{\Downarrow} h\, \overline{\pi} : \theta_1 \rho_r}\ \textsc{app-}\Downarrow$$

First, note that the $\vdash^{\mathsf{h}}$ and $\vdash^{\mathsf{inst}}$ relations are used in exactly the same way in both rules. Second, since the domain of θ_1 is the free instantiation variables of $\overline{\phi}$ and ρ_r, it must hold that:

$$\Theta = \mathsf{mgu}_{\mathsf{ql}}(\rho_r, \theta_1\rho_r) = \theta_1|_{\mathsf{fiv}(\rho_r)}$$

In particular we can split θ_1 in two independent substitutions, each covering a distinct domain of instantiation variables:

$$\theta_1 = \theta_1|_{\mathsf{fiv}(\overline{\phi})\setminus\mathsf{fiv}(\rho_r)} \cdot \theta_1|_{\mathsf{fiv}(\rho_r)}$$

Choose θ_2 in rule APP-$\Downarrow$ to be $\theta_1|_{\mathsf{fiv}(\overline{\phi})\setminus\mathsf{fiv}(\rho_r)}$. Then we have that $\theta_2 \cdot \Theta = \theta_1$. In particular, the last premise, which typechecks the arguments recursively, pushes the same types in both rules. $(\theta_2\Theta\phi_i = \theta_1\phi_i)$ □

C.2 Soundness of Type Inference (Section 7.2)

We give a set of auxiliary lemmas necessary to prove the main soundness results. We will be assuming that $\mathsf{fiv}(\Gamma) = \emptyset$, that is, the environment can only contain unification variables but not instantiation variables. That is certainly true and preserved during constraint generation.

Lemma C.2. *If* $\mathsf{mgu}_{\mathsf{ql}}(\rho_1, \rho_2) = \Theta$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\mathsf{mgu}_{\mathsf{ql}}(\theta\rho_1, \theta\rho_2) = (\theta\cdot\Theta)|_{\mathsf{dom}(\Theta)}$.

Proof. By definition of $\mathsf{mgu}_{\mathsf{ql}}$, we have that $\Theta\rho_1 = \Theta\rho_2$, and for any other Θ' such that $\Theta'\rho_1 = \Theta'\rho_2$ there exists $\Theta^\star$ such that $\Theta' = \Theta^\star \cdot \Theta$. Note that the domain of Θ, Θ', and $\Theta^\star$ are the instantiation variables of ρ_1 and ρ_2.

We have to check now that $(\theta\cdot\Theta)|_{\mathsf{dom}(\Theta)}$ satisfies the conditions to be $\mathsf{mgu}_{\mathsf{ql}}(\theta\rho_1, \theta\rho_2)$. First:

$$(\theta\cdot\Theta)|_{\mathsf{dom}(\Theta)}\ \theta\rho_i = \theta(\Theta(\theta\rho_i))$$

since ρ_i only contains free instantiation variables, which is exactly $\mathsf{dom}(\Theta)$. We can swap θ and Θ because they refer to disjoint sets of variables:

$$\theta(\Theta(\theta\rho_i)) = \theta(\theta(\Theta\rho_i))$$

Since $\Theta\rho_1 = \Theta\rho_2$, we have the desired equality.

To prove that $\theta\Theta$ is the most general, chose any other Θ'. If we repeat this process we reach:

$$\theta(\theta(\Theta'\rho_i))$$

Appealing to the fact that Θ is the most general unifier of ρ_i, we can rewrite that expression as:

$$\theta(\theta(\Theta^\star(\Theta\rho_i)))$$

By swapping θ to the deepmost position again, we see that $(\theta\cdot\Theta)|_{\mathsf{dom}(\Theta)}$ is indeed the $\mathsf{mgu}_{\mathsf{ql}}(\theta\rho_1, \theta\rho_2)$ □

Lemma C.3. *If* $\Gamma \vdash^{\mathsf{h}}_{\not{}} h : \sigma$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash^{\mathsf{h}}_{\not{}} h : \theta\sigma$.

Proof. Easy case analysis. □

Lemma C.4.

(1) If $\Gamma \vdash^{\mathsf{inst}} \sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\phi}\ ;\ \rho_r$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\Gamma \vdash^{\mathsf{inst}} \theta\sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\theta\phi}\ ;\ \theta\rho_r$.

(2) If $\Gamma \vdash_{\not{}} e : \phi \rightsquigarrow \Theta$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash_{\not{}} e : \theta\phi \rightsquigarrow \theta\cdot\Theta|_{\mathsf{fiv}(\phi)}$.

Proof. By mutual induction on the size of the term.

(1) Straightforward, by noticing that the substitution θ may not modify any of the instantiation variables generated during instantiation.
(2) Straightforward, by Lemma C.3, (1), and Lemma C.2.

□

As notational convenience, we write below $\theta \stackrel{\bullet}{\models} C$ to mean that $\mathsf{fiv}(\theta) = \emptyset$ and $\theta \models C$.

Lemma C.5. *If* $\Gamma \vdash^{\mathsf{inst}} \sigma \ ; \ \overline{\pi} \rightsquigarrow \overline{\phi} \ ; \ \rho_r \ ; \ C$ *and* $\theta \stackrel{\bullet}{\models} C$ *then* $\Gamma \vdash^{\mathsf{inst}} \theta\sigma \ ; \ \overline{\pi} \rightsquigarrow \overline{\theta\phi} \ ; \ \theta\rho_r$.

Proof. The only interesting case are rule ivarm, where we have to apply rule iarg in the declarative specification; and rule iarg, where we need to apply Lemma C.4. The rest of the cases go through by directly invoking the induction hypothesis or are trivial. □

Lemma C.6.

(1) If $\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \rightsquigarrow C$ *and* $\theta \stackrel{\bullet}{\models} C$ *then* $\theta\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \theta\sigma$.

(2) If $\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C$ *and* $\theta \stackrel{\bullet}{\models} C$ *then* $\theta\Gamma \vdash^{\forall}_{\Downarrow} e : \theta\sigma$.

(3) If $\Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow C$ *and* $\theta \stackrel{\bullet}{\models} C$ *then* $\theta\Gamma \vdash_{\Uparrow} e : \theta\rho$.

(4) If $\Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow C$ *and* $\theta \stackrel{\bullet}{\models} C$ *then* $\theta\Gamma \vdash_{\Downarrow} e : \theta\rho$.

Proof. By mutual induction on the size of the term. □

Theorem C.7 (Soundness). *If* $\Gamma \vdash_{\delta} e : \rho \rightsquigarrow C$, θ *is a substitution from unification variables to monotypes,* $\mathsf{fiv}(\theta) = \emptyset$, *and* $\theta \models C$ *then* $\theta\Gamma \vdash_{\delta} e : \theta\rho$.

Proof. Follows directly from Lemma C.6. □