A Quick Look at Impredicativity

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

Abstract

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.

Keywords Type systems, impredicative polymorphism, constraint-based inference

1 Introduction

Parametric polymorphism backed by Damas-Milner type inference took the world by storm when it was first introduced in ML [14]. But despite its enormous impact and adoption, 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 $\forall a.[a] \to [a]$ is fine (it is the type of the list reverse function), but the type $[\forall a.a \to a]$ 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.

Thus motivated, a long succession of papers have tackled the problem of type inference for impredicativity [1, 11, 12, 24, 27, 28]. 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 type system extensions, and implementable without an invasive rewrite of a type inference engine tailored to predicative type inference.

Until now: finally we have it. In this paper we introduce Quick Look, a new inference algorithm for impredicativity (Section 2) that (a) handles all the "obvious" examples (Figure 8); (b) is expressive enough to handle all of System F; (c) requires no extension to types, constraints, or intermediate representation; (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 3), and give the changes required for Quick Look (Section 4). 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.
- 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. Appendix Bgives the details for a much richer set of features.
- 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. Moreover, GHC uses a statically typed intermediate language based closely on System F; Quick Look allows source programs to be elaborated into this same intermediate language, rather than requiring (as does MLF [1]) a more sophisticated intermediate language.
- To better support impredicativity, we propose to abandon contravariance of the function arrow (Section 4.4). There are independent reasons for making this change [17], but it is illuminating to see how it helps impredicativity. Moreover, we provide data on its impact (Section 7).

In submission,
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

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

Figure 1. Type signatures for functions used in the text

- We prove a number of theorems about our system, including about which transformations do, and do not, preserve typeablity (Section 5).
- We give a type inference algorithm for Quick Look, in Appendix Afor space reasons. This algorithm is based on the now-standard approach of first *generating typing constraints* and then *solving them* [21]. As in the case of the declarative specification, no new forms of types or constraints are needed. Section 6 proves its soundness and completeness compared with the declarative specification in Section 4. The implementation is in turn based very closely on this algorithm.
- Appendix A and B also appear to be the first formal account of the extremely effective combination of bidirectional type inference [18] with constraint-based type inference [21, 26],

We cover the rich related work in Section 8.

2 The Key Ideas

Why not simply allow first-class polymorphism, so that $[\forall a.a \to a]$ is a valid type? The problem is in *type inference*. Consider the expression $(\mathit{single}\ \mathit{id})$, 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 $(\mathit{single}\ \mathit{id})$ 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 $(\mathit{head}\ \mathit{ids})$ 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 goal, just as in much previous work [24], is to exploit that special case.

2.1 The Quick Look

Our new approach works as follows:

- Treat applications as a whole: a function applied to a list of arguments.
- When instantiating the function, take a "quick look" at the arguments to guide that instantiation.
- If the quick look produces a definite answer, use it; if it produces no answer, instantiate with a monotype (as usual in Hindley-Damas-Milner type inference).

In our example $(\mathit{head}\ \mathit{ids})$, we have to instantiate the type of $\mathit{head} :: \forall p.[p] \to p$. The argument $\mathit{ids} :: [\forall a.a \to a]$ 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 $(\mathit{single}\ \mathit{id})$, a quick look sees that the argument $\mathit{id} :: \forall a.a \to a$ 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 the quick look produces no advice, and we revert to standard Hindley-Damas-Milner type inference by instantiating p with a monotype $\tau \to \tau$. (More operationally, the inference algorithm will instantiate p with a unification variable.)

Why is $(\mathit{head}\ \mathit{ids})$ 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 [24].

The quick-look approach scales nicely to handle multiple arguments. For example, consider the expression $(\mathit{id} : \mathit{ids})$, where (:) is Haskell's infix cons operator. How should we instantiate the type of (:) given in Figure 1? A quick look at the first argument, *id*, yields no information; it's like the $(\mathit{single}\ \mathit{id})$ case. But a quick look at the second argument, *ids*, 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.

2.2 "Quick-looking" 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:

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

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* [18, 19] as Section 3 discusses.

Taking a quick look at the result type is particularly important for lone variables. We can treat a lone variable as a degenerate form 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 $[\,] :: \forall p.[p]$. Now 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 $([\,] +\!\!+ ids)$. 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 $[\,]$.

2.3 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 the algorithm has 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. An easy (albeit ad-hoc) fix is to use a simple depth bound, because it is always safe to return no advice.

2.4 Interim Summary

We have described Quick Look at a high level. Its most prominent features are:

- A new "quick-look" phase is introduced, which guides instantiation of a call based on the context of the call: its arguments and expected result type.
- The quick-look phase is a modular addition. It guides at call sites, but the entire inference algorithm is otherwise undisturbed. That is in sharp contrast to 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.

3 Bidirectional, Higher-Rank Inference

We begin our formalisation by giving a solid baseline, closely based on *Practical type inference for arbitrary-rank types* [18], 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 4.4. We handle function application in an unusual way, one that will extend nicely for Quick Look, and we add *visible type application* [6].

3.1 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 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 4.5).

Similar to other works on type inference [4, 11, 27] 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 $\pi_1 \ldots \pi_n$.

Type constructors		$\ni$	F, G, T, ...	Includes ($\rightarrow$)
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	::=	$\tau \mid \mathsf{T}\,\overline{\sigma}$	
Fully mono. types	τ	::=	$\kappa \mid a \mid \mathsf{T}\,\overline{\tau}$	
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
Match flag	ω	::=	ma $\mid$ mnv	Match-any and match-non-var resp.
Environments	Γ	::=	$\epsilon \mid \Gamma, x:\sigma$	
Instantiation sets	Δ	::=	$\epsilon \mid \Delta, \kappa$	

Mono-substitutions	θ, ψ	::=	$[\overline{\alpha := \tau}]$	$\mathsf{fiv}(\sigma)$	Free instantiation variables of σ
Poly-substitutions	Θ, Φ	::=	$\bullet \mid \Psi$	$\mathsf{dom}(\theta)$	Domain of θ
	Ψ	::=	$[\overline{\kappa := \sigma}]$	$\mathsf{rng}(\theta)$	Range of θ
				$\Delta_1 \# \Delta_2$	Δ_1 is disjoint from Δ_2

Figure 2. Syntax

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.

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 3.4).

Note also that a lone variable x is a valid expression e; it is just an n-ary application with no arguments.

3.2 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 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 $\tau \rightarrow \rho$. 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.

Note also that, 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 will lead to a valid typing derivation.

Rule ABS-$\Downarrow$ handles a lambda in checking mode. The type being pushed down must be a function type $\sigma_a \rightarrow \sigma_r$; 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

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

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

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

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

$$\frac{\Gamma, x{:}\tau \vdash_{\Uparrow} e : \rho}{\Gamma \vdash_{\Uparrow} \lambda x.\, e : \tau \to \rho}\ \textsc{abs-}\Uparrow$$

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

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

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

$$\frac{\Gamma \vdash_{\Uparrow} e : \rho}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \rho}\ \textsc{h-infer}$$

Figure 3. Base type system: expressions

$$\boxed{\vdash^{\mathsf{inst}} \sigma[\overline{\pi}] \rightsquigarrow \Delta\ ;\ \overline{\phi}, \rho_r \quad \text{Invariants: } \Delta = \mathsf{fiv}(\overline{\phi}, \rho_r)}$$

$$\frac{\emptyset \vdash^{i} \sigma[\overline{\pi}] \rightsquigarrow \Delta\ ;\ \theta\ ;\ \overline{\phi}, \rho_r}{\vdash^{\mathsf{inst}} \sigma[\overline{\pi}] \rightsquigarrow (\Delta \setminus \mathsf{dom}(\theta))\ ;\ \overline{\theta\phi}, \rho_r}\ \textsc{inst}$$

$$\boxed{\begin{array}{c}\Delta \vdash^{i} \sigma[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r \\ \text{Invariants: } \mathsf{dom}(\theta)\#\Delta',\ \mathsf{dom}(\theta) \cup \Delta' = \mathsf{fiv}(\overline{\phi}, \rho_r) \\ \mathsf{fiv}(\mathsf{rng}(\theta)) \subseteq \Delta',\ \mathsf{dom}(\theta)\#\mathsf{fiv}(\rho_r)\end{array}}$$

$$\frac{}{\Delta \vdash^{i} \rho_r[] \rightsquigarrow \Delta\ ;\ \emptyset\ ;\ \rho_r}\ \textsc{iresult}$$

$$\frac{\begin{array}{c}(\overline{\pi} = \epsilon \ \text{ or } \ \overline{\pi} = e\,\overline{\pi'}) \quad \kappa \text{ fresh} \\ \Delta, \kappa \vdash^{i} ([a := \kappa]\rho)[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r\end{array}}{\Delta \vdash^{i} (\forall a.\, \rho)[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r}\ \textsc{iall}$$

$$\frac{\Delta \vdash^{i} (\rho[a := \sigma])[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r}{\Delta \vdash^{i} (\forall a.\rho)[\sigma\,\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r}\ \textsc{ityarg}$$

$$\frac{\Delta \vdash^{i} \sigma_2[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r}{\Delta \vdash^{i} (\sigma_1 \to \sigma_2)[e_1\,\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \sigma_1, \overline{\phi}, \rho_r}\ \textsc{iarg}$$

$$\frac{\begin{array}{c}\kappa \in \Delta \quad \mu, \upsilon \text{ fresh} \quad \theta_\kappa = [\kappa := (\mu \to \upsilon)] \\ (\Delta \setminus \kappa), \mu, \upsilon \vdash^{i} \upsilon[\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta\ ;\ \overline{\phi}, \rho_r\end{array}}{\Delta \vdash^{i} \kappa[e_1\,\overline{\pi}] \rightsquigarrow \Delta'\ ;\ \theta \circ \theta_\kappa\ ;\ \mu, \overline{\phi}, \rho_r}\ \textsc{ivard}$$

Figure 4. Base instantiation

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.

3.3 Applications, Instantiation, and Subsumption

A function application with n arguments (including $n = 0$) is dealt with by rule APP, whose four 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 σ with fresh *instantiation variables*, $\kappa, \mu, \ldots$, guided by the arguments $\overline{\pi}$ to which it is applied, using the judgment $\vdash^{\mathsf{inst}}$. The $\vdash^{\mathsf{inst}}$ judgment returns type ϕ_i for each of the value arguments, the top-level-monomorphic result type ρ_r of the call, and the *instantiation set* Δ of the freshly-instantiated variables.
3. Conjure up a "magic substitution" θ that maps all the instantiation variables in Δ to monotypes. At this point, the instantiation variables have completely disappeared.
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 (instantiated) 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 $\rho = \theta\phi_r$

The instantiation judgment in Figure 4

$$\vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta\ ;\ \overline{\phi}, \rho_r$$

implements step (2) by instantiating σ, guided by the arguments $\pi_1 \ldots \pi_n$. The type σ and expressions $\overline{e}$ should be

considered inputs; the instantiation set Δ, argument types $\overline{\phi}$, and result type ρ_r are outputs.

During instantiation, rule IALL instantiates a leading $\forall$, while rule IARG decomposes a function arrow. We discuss ITYARG in Section 3.4. When the argument list is empty, IRESULT simply returns the result type ρ_r. Note that the rules deal correctly function types that have a forall nested to the right of an arrow, e.g. $f :: Int \to \forall a.[a] \to [a]$.

Somewhat unusually, the instantiation judgement returns an *instantiation set* Δ, the set of fresh instantiation variables it created. The set Δ starts off empty, is augmented in IALL, and returned by IRESULT. Then after instantiation, in step (3), rule APP conjures up a monomorphic substitution θ for the instantiation variables in Δ, and applies it to $\overline{\phi}$ and ϕ_r. Just like the τ in ABS-$\Uparrow$, this θ comes "out of thin air".

You may wonder why we did not simply instantiate with arbitrary monotypes in the first place, and dispense with instantiation variables, and with θ. That would be simpler, but dividing the process in two will allow us to inject Quick Look between steps 2 and 3.

Finally, rule IVARD deals with the case that the function type ends in a type variable, but there is another argument to come. For example, consider ($id\ inc$ 3). We instantiate id with κ, so ($id\ inc$) has type κ; that appears applied to 3, so we learn that κ must be $\mu \to \upsilon$. We express that knowledge by extending a local substitution θ, and adding μ, υ to the instantiation set Δ. So we have

$$\vdash^{\mathsf{inst}} (\forall a.a \to a)[inc, 3] \rightsquigarrow \{\mu, \upsilon\} \ ; \ (\mu \to \upsilon), \mu, \upsilon$$

3.4 Visible Type Application

The $\vdash^{\mathsf{inst}}$ judgement also implements visible type application (VTA) [6], 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. The argument is a polytype σ: we allow impredicative instantiation. Note that we do not address all the details of the design of Eisenberg et al. [6]. In particular, we do not account for the difference between "specified"' and "inferred" quantifiers.

The attentive reader will 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. [5], but the plumbing is distracting so we omit it.

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

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \quad \overline{e} = \mathsf{valargs}(\overline{\pi}) \\ \vdash^{\mathsf{inst}} \sigma[\overline{\pi}] \rightsquigarrow \Delta \ ; \ \overline{\phi}, \rho_r \\ \overline{\Gamma \vdash_{\not\downarrow} e_i : (\Delta, \phi_i) \rightsquigarrow \Theta_i} \\ \Theta_r = \begin{cases} \emptyset & \text{if } \delta = \Uparrow \\ \mathsf{match}(\Delta, \rho_r, \rho, \mathsf{mnv}) & \text{if } \delta = \Downarrow \end{cases} \\ \Theta = (\oplus \Theta_i) \oplus \Theta_r \quad \overline{\phi'_i = \Theta\phi_i} \quad \forall \overline{a}.\rho'_r = \Theta\rho_r \\ \mathsf{dom}(\theta) = \overline{a} \cup (\Delta \setminus \mathsf{dom}(\Theta)) \\ \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi'_i} \quad \rho = \theta\rho'_r \end{array}}{\Gamma \vdash_{\delta} h\,\overline{\pi} : \rho}\ \textsc{app}$$

$$\boxed{\begin{array}{c}\text{QL argument} \quad \Gamma \vdash_{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \Theta \\ \text{Invariants: } \mathsf{dom}(\Theta) \subseteq \Delta \subseteq \mathsf{fiv}(\phi)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\not\downarrow} h : \sigma \quad \vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta_\phi \ ; \ \overline{\phi}, \rho_r \\ \overline{\Gamma \vdash_{\not\downarrow} e_i : (\Delta_\phi, \phi_i) \rightsquigarrow \Theta_i} \quad \Theta_\phi = \oplus \Theta_i \\ \omega = \text{if } \Delta_\phi = \emptyset \text{ then ma else mnv} \end{array}}{\Gamma \vdash_{\not\downarrow} h\,\overline{e} : (\Delta_\rho, \rho) \rightsquigarrow \mathsf{match}(\Delta_\rho, \rho, \Theta_\phi\rho_r, \omega)}\ \textsc{app-}\not\downarrow$$

$$\frac{e \text{ not an app. or } \phi \text{ not top-level mono.}}{\Gamma \vdash_{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \emptyset}\ \textsc{rest-}\not\downarrow$$

$$\boxed{\text{QL head} \quad \Gamma \vdash^{\mathsf{h}}_{\not\downarrow} h : \sigma}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}}_{\not\downarrow} x : \sigma}\ \textsc{h-var-}\not\downarrow \qquad \frac{}{\Gamma \vdash^{\mathsf{h}}_{\not\downarrow} (e :: \sigma) : \sigma}\ \textsc{h-ann-}\not\downarrow$$

Figure 5. Quick look impredicativity, part 1

4 Quick Look Impredicativity

Building on the baseline of Section 3, we are now ready to present Quick Look, the main contribution of this paper. The modified typing rules appear in Figure 5, supported by new judgments in Figure 6.

The sole change to the typing rules for expressions is the shaded portion of rule APP (Figure 5). We take a quick look at each argument, and at the result type, each of which returns a substitution Θ; then we combine all those Θ's with $\oplus$, and apply the resulting substitution to the argument types $\overline{\phi}$ and result type ρ_r.

A substitution Θ maps instantiation variables (in Δ) to *polytypes* (Figure 2). It embodies the information deduced from each argument, and from the result type. It is always sound for Θ to be empty, meaning "no useful quick-look information".

$$\boxed{\mathsf{match}(\Delta, \rho_1, \rho_2, \omega) = \Theta}$$

$$\begin{array}{lcl}
\mathsf{match}(\Delta, \rho_1, \rho_2, \omega) & & \\
\quad = & \emptyset & \text{if } \omega = \mathsf{mnv},\ \rho_1 = \kappa \in \Delta \\
\quad = & \Psi & \text{if } \Psi\rho_1 = \rho_2,\ \mathsf{dom}(\Psi) \subseteq \Delta \\
\quad = & \bullet & \text{otherwise}
\end{array}$$

$$\boxed{\Theta_1 \oplus \Theta_2 = \Theta_3}$$

$$\begin{array}{rcl}
\bullet \oplus \Theta_2 & = & \bullet \\
\Theta_1 \oplus \bullet & = & \bullet \\
\Psi_1 \oplus \Psi_2 & = & [\ \kappa := \mathsf{join}(\Psi_1\kappa, \Psi_2\kappa) \\
 & & \ \mid \kappa \in \mathsf{dom}(\Psi_1) \cup \mathsf{dom}(\Psi_r)\ ] \\
 & & \text{if all those joins are defined} \\
 & = & \bullet \quad \text{otherwise}
\end{array}$$

$$\boxed{\mathsf{join}(\sigma_1, \sigma_2) = \sigma_3}$$

join is undefined on some arguments

e.g. join(*Maybe Int*, *Maybe Bool*) is undefined

$$\begin{array}{rcll}
\mathsf{join}(\mathsf{T}\,\overline{\sigma}, \mathsf{T}\,\overline{\phi}) & = & \mathsf{T}\,\overline{\mathsf{join}(\sigma_i, \phi_i)} & \\
\mathsf{join}(\forall a.\sigma_1, \forall a.\sigma_2) & = & \forall a.\mathsf{join}(\sigma_1, \sigma_2) & \\
\mathsf{join}(a, a) & = & a & \\
\mathsf{join}(\kappa, \kappa) & = & \kappa & \\
\mathsf{join}(\kappa, \mu) & = & \upsilon & \kappa \neq \mu,\ \upsilon \text{ fresh} \\
\mathsf{join}(\kappa, \sigma_2) & = & \sigma_2 & \\
\mathsf{join}(\sigma_1, \kappa) & = & \sigma_1 & \\
\multicolumn{4}{c}{\mathsf{join} \text{ is undefined on other arguments}}
\end{array}$$

Figure 6. Quick look impredicativity, part 2

4.1 A Quick Look at the Argument

This judgement takes a quick look at an argument e:

$$\Gamma \vdash_{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \Theta$$

Here e is the argument, ϕ is the type expected by the function, and Θ is advice (in the form of a substitution) gained from comparing e with ϕ. The rules of this judgement are given in Figure 5. Rule REST-$\not\downarrow$ is the base case for $\vdash_{\not\downarrow}$. It does nothing, returning the empty substitution, for any complicated argument: list comprehensions, case expressions, lambdas, etc etc. Only in the case of a function application $h\,\overline{e}$ (including the case of a lone variable) does $\vdash_{\not\downarrow}$ do anything at all (Section 2.3).

Looking now at rule APP-$\not\downarrow$, when the argument is itself a call ($h\,\overline{e}$) we use the same judgement again, recursively. This is perhaps confusing at first, so here is APP-$\not\downarrow$ specialised to a lone variable x:

$$\frac{x : \forall \overline{a}.\rho \in \Gamma \quad \Delta_\phi = \overline{\kappa}\ \text{fresh} \quad \rho_r = [\overline{a := \kappa}]\rho \quad \omega = \text{if } \Delta_\phi = \emptyset \text{ then ma else mnv}}{\Gamma \vdash_{\not\downarrow} x : (\Delta, \rho) \rightsquigarrow \mathsf{match}(\Delta, \rho, \rho_r, \omega)}$$

We look up x in Γ, instantiate any top-level quantifiers, and then use *quick-look matching*, $\mathsf{match}(\Delta, \rho, \rho_r, \omega)$, to find a substitution Θ, over variables in Δ, that makes ρ look like ρ_r.

The quick-look matching function $\mathsf{match}(\Delta, \rho_1, \rho_2, \omega)$, defined in Figure 6, returns a poly-substitution Θ (Figure 2). We discuss the first equation in Section 4.2. The second equation returns a subsitution Ψ from variables in Δ to polytypes σ, if $\Psi\rho_1 = \rho_2$. If there is no such substition we have a type error, and we return $\Theta = \bullet$.

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 match *head*'s expected argument type $[\kappa]$ against it. That yields $\Theta = [\kappa := \forall a.a \to a]$, as desired.

4.2 Naked variables

What about the first equation of match, and the mysterious parameter ω? Consider the call (*single id*). The argument type of $single :: \forall p.p \to [p]$ is a bare variable; after instantiation it will be, say, κ. The actual argument 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, the first equation of match bales out, returning an empty substitution $\emptyset$. Perhaps one of the other arguments of the application will force an unambiguous choice for κ, as indeed happens in the case of ($id : ids$).

This behaviour is controlled by match's fourth parameter ω. If $\omega = \mathsf{mnv}$ – short for "match non-variable" – and ρ_1 is a naked variable κ, match returns the empty substitution. When do we need to trigger this behaviour? Answer: when the argument is polymorphic, expressed by setting $\omega = \mathsf{mnv}$ if $\Delta_\phi \neq \emptyset$ in APP-$\not\downarrow$.

If, on the other hand, there is no instantiation in the argument ($\Delta_\phi = \emptyset$), APP-$\not\downarrow$ sets $\omega = \mathsf{ma}$, indicating that match can proceed regardless of whether ρ_1 is a naked variable. For example, in the call (*single ids*) we will perform the quick-look match

$$\mathsf{match}(\{\kappa\}, \kappa, [\forall a.a \to a], \mathsf{ma})$$

Here $\omega = \mathsf{ma}$ disables the first equation of match, so the match returns $\Theta = [\kappa := [\forall a.a \to a]]$.

A very similar issue arise in rule APP (Figure 5). In checking mode ($\delta = \Downarrow$), we compute Θ_r by doing a quick-look match between the expected result type ρ and the actual result type ρ_r. For example, consider the term

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

We instantiate *single* with κ, but (as we saw at the beginning of this subsection) we fail to learn anything from a quick look at the first argument. However, the call to $\mathsf{match}(\Delta, \rho_r, \rho, \mathsf{mnv})$ in rule APP matches the result type $\rho_r = [\kappa]$ with the type

expected by the context $\rho = [\forall a.a \to a]$, yielding $\Theta_r = [\kappa := \forall a.a \to a]$, as desired.

Why $\omega = \mathsf{mnv}$ in this call? Consider $(\mathit{head}\ \mathit{ids}) :: \mathit{Int} \to \mathit{Int}$. After instantiating *head* with κ, a quick look at the argument to *head* will give $\Theta_1 = [\kappa := \forall a.a \to a]$. But we will independently do a quick look at the argument

$$\Theta_r = \mathsf{match}(\{\kappa\}, \kappa, \mathit{Int} \to \mathit{Int}, \mathsf{mnv})$$

If we instead used $\omega = \mathsf{ma}$ we would compute $\Theta_r = [\kappa := \mathit{Int} \to \mathit{Int}]$, contradicting Θ_1. The right thing to do is to use $\omega = \mathsf{mnv}$, which declines to quick-look match if the result type ρ_r is a naked variable.

4.3 Some Tricky Corners

In this section we draw attention to some subtler details of the typing rules.

Quick look does not replace "proper" typechecking. In rule APP, we take a quick look at the arguments, with $\vdash^{\mathsf{inst}}$, but we *also* do a proper typecheck of each argument in the final premise, using $\vdash^{\forall}_{\Downarrow} e_i : \phi_i$. This modular separation is what allows the quick-look part such leeway, because once the instantiation is chosen, the arguments will always be typechecked as normal.

Calls as arguments. In rule APP-$\not\downarrow$, we generalise the rule given in Section 4.1 to work for calls as well as for lone variables. We can do that very simply by recursively invoking $\vdash^{\mathsf{inst}}$. Our only goal is to discover the return type $\Theta_\phi \rho_r$ to use in the quick-look match in the conclusion. Rule APP-$\not\downarrow$ sets ω based on whether Δ_ϕ is empty, which neatly checks for instantiation *anywhere* in the argument. For example consider $(\mathit{single}\ (f\ 3))$ where $f :: \mathit{Int} \to \forall a.a \to a$, where the instantation is hidden under an arrow.

Instantiation after quick look. In rule APP while ρ_r is top-level monomorphic, $\Theta\rho_r$ may not be. For example, consider $(\mathit{head}\ \mathit{ids})$. Instantiating *head* with $[p := \kappa]$ gives $\rho_r = \kappa$. But a quick look at the argument gives $\Theta = [\kappa := \forall a.a \to a]$, so $\Theta\rho_r$ is $\forall a.a \to a$. Hence APP is careful to decompose $\Theta\rho_r$ to $\forall \overline{a}.\rho'_r$, and then includes $\overline{a}$ in the domain of the "magic substitution" θ.

Quick look binds only instantiation variables. In match (Figure 6), we find a substitution Θ that matches ρ_1 with ρ_2, *binding only instantiation variables in* Δ. The sole goal is to find the best type with which to instantiate the call; we leave all other variables unaffected.

Joining the results of independent quick looks. The operator $\oplus$, defined in Figure 6, uses a function join to combine the substitutions from independent quick looks. A tricky example is the call $(f\ x\ y)$ where

$$f :: \forall a.a \to a \to \mathit{Int}$$
$$x :: \forall b.(b, \sigma_{id}) \quad y :: \forall c.(\sigma_{id}, c)$$

and σ_{id} is shorthand for $\forall a.a \to a$. Suppose we instantiate f with $[a := \kappa]$. A quick look at the two arguments then yields

$$\Theta_1 = [\kappa := (\sigma_{id}, v_1)] \quad \Theta_2 = [\kappa := (v_2, \sigma_{id})]$$

where v_1, v_2 come from instantiating x and y respectively. Joining these leads to the desired final $\Theta = [\kappa := (\sigma_{id}, \sigma_{id})]$. The design of our join function is justified by the need to provide a sound and complete algorithm for type inference that respects this specification. (Lemma 6.3)

The fifth equation of join, returning a fresh v, is a technical device to ensure that $\oplus$ is commutative. Neither side gives any information.

4.4 Co- and Contravariance of Function Types

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

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

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 uniformly means that it acts as a guard, which in turn allows more impredicative instantiation to be inferred. For example, without an invariant function arrow $(\mathit{app}\ \mathit{runST}\ \mathit{argST})$ cannot be typed.

Second, such mismatches are rare (we give data in Section 7), 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" [18, §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 [15], 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. There is a current GHC Proposal to adopt invariant function arrow [17].

4.5 Modularity

Quick Look is like many other works in that it exploits programmer-supplied type annotations to guide type inference (Section 8). But Quick Look's truly distinctive feature is that it is *modular* and *highly localised*.

High localised. Through half-closed eyes the changes in Figures 5 and 6 seem substantial. Rule APP is the sole rule of the expression judgement that is 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. 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 [26]. *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, Appendix Bgives the extra rules for a much larger language; and we have built a full implementation in GHC (Section 7). 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.

5 Properties of Quick Look

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

5.1 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:

Theorem 5.1. *If* $\Gamma \vdash_{\Uparrow} e : \rho$ *then* $\Gamma \vdash_{\Downarrow} e : \rho$.

Although the statement of the theorem is easy, the inductive proof case for rule APP involves a delicate split on whether the QL from the arguments yields a polymorphic type or not. The subsequent QL on the result cannot cause any further instantiations.

5.2 Stability under Transformations

In any type system it is desirable that simple program transformations do not make a working program fail to typecheck, or vice versa. In this section we give some such properties, using a slightly larger language than the one we presented in the paper.

Argument permutation One obviously-desirable property is that a program should be insensitive to permutation of function arguments. And indeed that is true.

Theorem 5.2 (Argument permutation)**.** *Let* Γ *be an environment,* $\overline{e_i}$ *expressions. If:*

$$\Gamma, f : \forall \overline{a}.\, \phi_1 \to \cdots \to \phi_n \to \phi_r \vdash_\delta f\ \overline{e_i} : \sigma$$

then for any permutation π *of the indices* i *we have:*

$$\Gamma, f' : \forall \overline{a}.\, \phi_{\pi(1)} \to \cdots \to \phi_{\pi(n)} \to \phi_r \vdash_\delta f'\ \overline{e_{\pi(i)}} : \sigma$$

Proof. Both the quick-look and the type checking of arguments is done "in parallel" in rule APP. Furthermore, quick-looking is done separately for each argument, and then joined. Since the definition of join is independent of the order of the poly-substitutions, the theorem holds. □

Unfortunately, there is a trade-off here: arguments cannot choose whether or not to instantiate based on the quick look on other arguments. One particular case is the use of the application function $app :: \forall a\ b.(a \to b) \to a \to b$; the expression $app\ f\ x$ does not type check as often as the direct application $f\ x$ does.

Let abstraction and inlining One desirable property held by ML is let-abstraction:

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

This property does not hold in most higher-rank systems, including PTIAT, because they use the context of the call to guide 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; but it can be fixed by adding a type signature:

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

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

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

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

η-expansion. Sometimes, as we have seen in Section 4.4, η-expansion is necessary to make a program typecheck. But sometimes the reverse is the case. For example, whereas *app runST argST* is accepted, *app* $(\lambda x\,.\ runST\ x)$ *argST* is not. The problem is that the fact that *app* must be instantiated impredicatively comes solely from *runST*; *argST* alone is not enough to know whether $\forall s.ST\ s\ v$ must be left as it is or instantiated further.

5.3 Type soundness

For reasons of space, we do not explicitly address type soundness of the proposed type system. However, our algorithm – provably sound with respect to the declarative specification – elaborates to System F with explicit equality coercions [25], a provably-sound type system.

6 Type inference

So far we have given a declarative type system that explains when a program type checks, but we have not given a *type inference algorithm* that implements this specification. 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* [26], 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 Appendix Bdemonstrate.

Our algorithm, in Appendix A, presents judgements of the form $\Gamma \vdash_\delta e : \rho \rightsquigarrow C$ that generate constraints C. These judgements follow closely the judgements of the declarative specification, but instead of clairvoyantly selecting monomorphic types for λ-abstraction arguments and for other instantiations (e.g. the range of substitution θ in rule App) they create fresh *unification variables*.

Unification variables stand for monomorphic types, and are solved for by *constraint solving* of the C constraints; as opposed to instantiation variables that are solved for by Quick Look. Hence the following invariant:

Lemma 6.1. *If* $\Gamma \vdash_\delta e : \rho \rightsquigarrow C$ *(with* $\mathsf{fiv}(\Gamma) = \emptyset$*) then* $\mathsf{fiv}(C) = \emptyset$.

We write $\theta \models C$ to mean that a *monomorphic* idempotent substitution θ (from any sort of variables) is a solution to constraint C[1]. Due to Lemma 6.1, 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 6.2 (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* $\theta\Gamma \vdash_\delta e : \theta\rho$.

[1]For implication constraints θ is a substitution nesting – see [24].

The theorem relies on a chain of other lemmas for every auxiliary judgement used in our specification and the algorithm. One of the basic facts required for this chain of lemmas justifies our design of the $\oplus$ operator on substitutions and the join(ϕ_1, ϕ_2) function:

Lemma 6.3. *If* $\Theta^\star = \oplus\overline{\Theta}$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then*

$$(\theta\cdot\Theta^\star)|_{\mathsf{dom}(\Theta^\star)} = \oplus(\theta\cdot\Theta_i)|_{\mathsf{dom}(\Theta_i)}$$

Theorem 6.4 (Completeness)**.**

- *If* $\Gamma \vdash_{\Uparrow} e : \rho$ *then* $\Gamma \vdash_{\Uparrow} e : \rho' \rightsquigarrow C$ *and there exists a solution* $\psi \models C$ *with* $\mathsf{fiv}(\psi) = \emptyset$ *such that* $\psi\rho' = \rho$.
- *If* $\Gamma \vdash_{\Downarrow} e : \rho$ *then* $\Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow C$ *and there exists a solution* $\psi \models C$ *with* $\mathsf{fiv}(\psi) = \emptyset$.

The constraints C generated by our judgements admit principal solutions [21]; combining this fact with completeness implies that our type system admits principal types.

7 Implementation and Evaluation

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-8.9.0.20191101). The changes were straightforward, and were highly localised. Overall we added 800 lines, and removed 200 lines, from GHC's 90,000 line type inference engine (these figures include comment-only lines, since they are a good proxy for complexity). The implementation is publicly available.[2]

Adding Quick Look to instantiation in rule App cannot make any program fail to typecheck if it typechecked before. However, in Section 4.4 we propose to make the function arrow invariant, and to drop deep instantiation and deep skolemisation. This change is not fundamental to Quick Look, but it increases its usefulness. 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 7. 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.

[2]URL suppressed for review, but available on request.

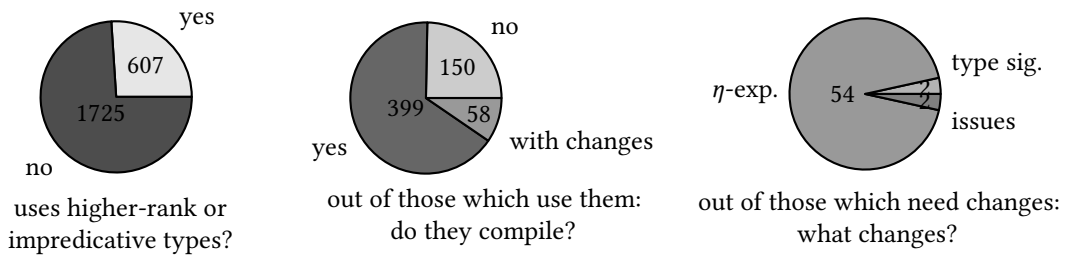

Figure 7. Summary of the evaluation, using Stackage LTS 13.6

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 was 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 also needed some local type signatures.

8 Related Work

This section explores many different strands of work on first-class polymorphism; another excellent review can be found in [2, §5]. Figure 8 compares the expressiveness of some of these systems, using examples culled from their papers. As you can 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 bidirectionality [18]. Follow-up modern presentations [3] re-frame the problem within a more logical setting, and additionally describe extensions to indexed types [4].

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 [16]. 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 Poly-ML [8]. 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 [7] the programmer chooses explicitly when to not instantiate a polymorphic type by using the freeze operator $\lceil - \rceil$. Another variant of this line of work is QML [23], 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. We urge the reader to consult the related work section in the latter work for an overview of that line of research.

Explicit wrapping and unwrapping is painful, especially since it often seems unnecessary, so our goal is to allow polymorphic functions to be implicitly instantiated with quantified types.

Guarded impredicativity. Quick Look builds on ideas originating in previous work on Guarded Impredicative Polymorphism (GI) [24]. 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 in GHC.

HMF [11] comes the closest to QL in terms of expressiveness, a simple type vocabulary, and an equation-based unification algorithm. The key idea in HMF is that in an n-ary application we perform a subsumption between each function argument type and the type of the corresponding

		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′*	✓	✓	✓	✓	✓	✓
A7	*choose id auto*	✓	✓	✓	No	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*	✓	Ext*	✓	No	✓	✓
	where *f* :: $\forall a.\ (a \to a) \to [a] \to a$						
	GI needs an annotation on *id* :: $(\forall a.\ a \to a) \to (\forall a.\ a \to a)$.						
A10	*poly id*	✓	✓	✓	✓	✓	✓
A11	*poly* ($\lambda x.\ x$)	✓	✓	✓	✓	✓	✓
A12	*id poly* ($\lambda x.\ x$)	✓	✓	✓	✓	✓	✓
B	INFERENCE OF POLYMORPHIC ARGUMENTS						
B1	$\lambda f.\ (f\ 1, f\ True)$	No	No	No	No	No	No
	All systems require an annotation on *f* :: $\forall a.\ a \to a$.						
B2	$\lambda xs.$ *poly* (*head xs*)	No	No	✓	No	No	No
	All systems except for MLF require annotated *xs* :: $[\forall a.\ a \to a]$.						
C	FUNCTIONS ON POLYMORPHIC LISTS						
C1	*length ids*	✓	✓	✓	✓	✓	✓
C2	*tail ids*	✓	✓	✓	✓	✓	✓
C3	*head ids*	✓	✓	✓	✓	✓	✓
C4	*single id*	✓	✓	✓	✓	✓	✓
C5	*id* : *ids*	✓	✓	✓	No	✓	✓
C6	($\lambda x.\ x$) : *ids*	✓	✓	✓	No	✓	✓
C7	*single inc* ⧺ *single id*	✓	✓	✓	✓	✓	✓
C8	*g* (*single id*) *ids*	✓	No	✓	No	✓	✓
	where *g* :: $\forall a.\ [a] \to [a] \to a$						
C9	*map poly* (*single id*)	✓	No	✓	✓	✓	✓
	GI needs an ann. on *single id* :: $[\forall a.\ a \to a]$ in the previous two.						
C10	*map head* (*single ids*)	✓	✓	✓	✓	✓	✓
D	APPLICATION FUNCTIONS						
D1	*app poly id*	✓	✓	✓	✓	✓	✓
D2	*revapp id poly*	✓	✓	✓	✓	✓	✓
D3	*runST argST*	✓	✓	✓	✓	✓	✓
D4	*app runST argST*	✓	✓	✓	✓	✓	✓
D5	*revapp argST runST*	✓	✓	✓	✓	✓	✓
E	η-EXPANSION						
E1	*k h lst*	No	No	No	No	No	No
E2	*k* ($\lambda x.$ *h x*) *lst*	✓	✓	✓	No	✓	✓
	where *h* :: $Int \to \forall a.\ a \to a$, *k* :: $\forall a.\ a \to [a] \to a$, and *lst* :: $[\forall a.\ Int \to a \to a]$						
E3	*r* ($\lambda x\ y.\ y$)	✓	Ext*	✓	No	No	No
	where *r* :: $(\forall a.\ a \to \forall b.\ b \to b) \to Int$						

* "Ext" in the GI column refers to programs which are not accepted by the vanilla system, but are allowed with some of its extensions [24].

Figure 8. Comparison of impredicative type systems

argument. The order in which to perform these subsumption checks is delicate: it is first performed for those arguments that correspond to an argument type that is guarded (i.e. not a naked type variable), and then on the other arguments in any order. This bears a strong similarity to our system and to GI.

A key difference is that HMF is formulated to *generalize eagerly* (and solve impredicativity in one go), whereas GHC does exactly the opposite: it defers generalisation as long as possible in favour of generating constraints, only performing constraint-solving and generalisation when absolutely necessary. Switching to eager generalisation would be a deep change to GHC's generate-and-solve approach to inference, and it is far from clear how it would work. On the other hand HMF comes with a high-level specification that enforces that polymorphism is not-guessed in the typing rules with a condition requiring that the types we assign to terms have minimal "polymorphic weights".

Another small technical difference is that our system explicitly keeps track of the variables it can unify to polytypes (with the Δ environments), whereas HMF allows arbitrary unification of any variable to a polytype when checking *n*-ary applications, accompanied with an after-the-fact check that no variable from the environment was unified to a polytype. This is merely a difference in the presentation and mechanics but not of fundamental nature.

Stratified inference. The idea of a "quick-look" as a restricted pass prior to actual type inference has appeared before in the work on Stratified Type Inference (STI) [20, 22]. 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 iterated [20, §7]. These works handle higher-rank types and GADTs, but to date there has been no STI design for impredicativity. QL has a similar flavor, but admittedly a more algorithmic specification – for example stratified type inference does not use substitutions – rather *joins* of types in the polymorphic lattice. Nor does it return substitutions *and* constraints. On the other hand, STI can only propagate explicitly *declared* types whereas, by being part of type inference, QL can exploit the *inferred* types for top-level or locally let-bound functions.

Beyond System F. We have been reluctant to move beyond System F types, because doing so impacts the language the programmers 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 [1], but there are several subsequent variants, including HML [12], and FPH [28].

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 without any further annotations without affecting typeability; a corollary of principal types.

However, MLF and variants do require intrusive modifications to a constraint solver (and in the case of GHC, a complex constraint solver with type class constraints, implication constraints, type families, and more) and to the type structure. Though some attempts have been made to integrate MLF with qualified types [13], a full integration is uncharted territory. Hence, we present here a pragmatic compromise in expressiveness for simplicity and ease of integration in the existing GHC type inference engine.

9 Conclusion and Future Work

In this paper we have presented Quick Look, a new approach to impredicative polymorphism which is both modular and highly localised. We have implemented the system within GHC, and evaluated the (scarce) changes required to a wide set of packages.

Quick Look focuses on inferring impredicative *instantiations*. We plan to investigate whether a similar approach could be used to infer polymorphic types in *argument* position. For example, being able to accept the term $(\lambda x\,.\,x : ids)$. As in the case of visible type application with impredicativity instantiation, we expect *type variables in patterns* [5] to help with argument types.

Acknowledgments

We thank Richard Eisenberg, Stephanie Weirich, and Edward Yang 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

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

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

[3] 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

[4] 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

[5] 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

[6] 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

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

[8] 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.

[9] 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).

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

[11] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism, See [10], 283–294. https://doi.org/10.1145/1411204.1411245

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

[13] 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

[14] 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

[15] 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

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

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

[18] 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.

[19] 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

[20] 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 (POPL '06)*. ACM, New York, NY, USA, 232–244. https://doi.org/10.1145/1111037.1111058

[21] 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/

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

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

[24] 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/

[25] Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, François Pottier and George C. Necula (Eds.). ACM, 53–66. https://doi.org/10.1145/1190315.1190324

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

[27] 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

[28] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell, See [10], 295–306. https://doi.org/10.1145/1411204.1411246

A Type Inference Algorithm

We have already sketched the basic design of a type inference algorithm in Section 5. Here we give more details.

The key difference between the declarative specification and an algorithmic formulation is that the latter introduces *unification variables*, denoted with α, β, γ, and *constraints* (see Figure 9). 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 *OutsideIn(X): Modular type inference with local assumptions* [26] (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.

The algorithm for generating constraints is given in Figures 10, 11, and 12. We highlight some key points.

Algorithm: expressions Figure 10 presents constraint generation for expressions. 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 $\overline{a}$.

Rules ABS-⇓-VAR and ABS-⇑ generate fresh unification variables, as expected. Note that they preserve the invariant that environments and constraints only mention unification variables but never instantiation variables.

Rule APP follows very closely its declarative counterpart in Figure 5, with a few minor deviations. First, while rule APP 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 10 generates fresh unification variables $\overline{\alpha}$. Second, rule APP in Figure 10 uses the $\equiv$ judgement to generate further constraints; whereas the rule in Figure 5 has readily ensured that $\rho = \theta\rho_r$.

Algorithm: instantiation The algorithmic instantiation judgement in Figure 11 collects up constraints that are generated in rule IVARM. In that case we meet a unification variable α that we have to further unify to a function type $\beta \to \gamma$. Note how the constraint C constraint only mentions unification *but no instantiation variables*. Instantiation variables κ are born and eliminated in a single quick look.

Algorithm: quick look Constraint generation for the Quick Look argument and Quick Look subsumption relations in Figure 12 is fairly unsurprising – simply generate and collect constraints. However QL matching is delicately different than its declarative version in Figure 6, and specifically rule U-GD. While the declarative version only attempts to do matching, the algorithmic does unification in order to account for the presence of ordinary unification variables. It then splits the most-general unifier into a substitution on instantiation variables, and a set of constraints, mentioning

only unification variables, in order to preserve our invariant about constraints not mentioning instantiation variables.

Properties: basics, soundness, completeness Section 5 already presents several properties of our constraint generation judgements. Here we give a set of auxiliary lemmas necessary to prove the main 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 A.1. *If* $\Gamma \vdash^{\mathsf{h}}_{\not\Downarrow} h : \sigma$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash^{\mathsf{h}}_{\not\Downarrow} h : \theta\sigma$.

Lemma A.2. *If* $\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \rightsquigarrow C$ *and* $\theta \models C$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \theta\sigma$.

Lemma A.3. *If* $\vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta \; ; \; \overline{\phi}, \rho_r \; ; \; C$ *and* $\theta \models C$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then:* $\vdash^{\mathsf{inst}} \theta\sigma[\overline{e}] \rightsquigarrow \Delta \; ; \; \overline{\theta\phi}, \theta\rho_r$.

Lemma A.4. *If* $\Gamma \vdash_{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \Theta \; ; \; C$, $\mathsf{fiv}(\theta) = \emptyset$ *and* $\theta \models C$ *then* $\theta\Gamma \vdash_{\not\downarrow} e : (\Delta, \theta\phi) \rightsquigarrow \theta\cdot\Theta|_\Delta$

Lemma A.5. *If* $\vdash^{\not\downarrow}_{\delta} (\Delta, \rho_1) \sqsubseteq \rho_2 \rightsquigarrow \Theta; C$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *and* $\theta \models C$ *then* $\vdash^{\not\downarrow}_{\delta} (\Delta, \theta\rho_1) \sqsubseteq \theta\rho_2 \rightsquigarrow (\theta\cdot\Theta)|_\Delta$.

Proof. (Sketch) This proof is actually interesting hence we sketch it here. Let's denote the result of splitting as $\theta_{\neg\Delta}$ and Θ_Δ. Hence $\theta_{\neg\Delta} \cup \Theta_\Delta = mgu(\rho_1, \rho_2)$. The returned constraint C is just the $\theta_{\neg\Delta}$ set of constraints. Moreover we know that $\theta \models \theta_{\neg\Delta}$. Because this is a unifier we know that: $\theta_{\neg\Delta}\Theta_\Delta(\rho_1) = \theta_{\neg\Delta}(\rho_2)$ but because the two substitutions are independent and idempotent we can apply them in any order, hence: $\Theta_\Delta(\theta_{\neg\Delta}(\rho_1)) = \theta_{\neg\Delta}(\rho_2)$. That in turn means that: $\vdash^{\not\downarrow}_{\delta} (\Delta, \theta_{\neg\Delta}(\rho_1)) \sqsubseteq \theta_{\neg\Delta}(\rho_2) \rightsquigarrow \Theta_\Delta$ (because matching will return the most general substitution, which must be Θ_Δ). But, because $\theta \models \theta_{\neg\Delta}$ it must be that: $\theta = \psi \cdot \theta_{\neg\Delta}$ for some ψ. Hence we get: $\vdash^{\not\downarrow}_{\delta} (\Delta, \psi\theta_{\neg\Delta}(\rho_1)) \sqsubseteq \psi\theta_{\neg\Delta}(\rho_2) \rightsquigarrow \psi\Theta_\Delta$ (by an easy substitution property of the declarative specification). However $\psi\Theta_\Delta$ restricted to Δ is equal to $\psi\theta_{\neg\Delta}\Theta_\Delta$ on Δ, and that is exactly $\theta\Theta_\Delta$ (restricted to Δ) as required. □

Lemma A.6. *If* $\vdash_\delta \rho_1 \equiv \rho_2 \rightsquigarrow C$ *and* $\theta \models C$ *then* $\theta\rho_1 = \theta\rho_2$.

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

Lemma A.7 (Soundness)**.** *The following are true (proof by mutual induction on the size of the term):*

- *If* $\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C$ *and* $\theta \models\!\!\!\bullet C$ *then* $\theta\Gamma \vdash^{\forall}_{\Downarrow} e : \theta\sigma$.
- *If* $\Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow C$ *and* $\theta \models\!\!\!\bullet C$ *then* $\theta\Gamma \vdash_{\Uparrow} e : \theta\rho$.
- *If* $\Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow C$ *and* $\theta \models\!\!\!\bullet C$ *then* $\theta\Gamma \vdash_{\Downarrow} e : \theta\rho$.

Theorem 6.2 follows directly from Lemma A.7.

Finally, the following is an auxiliary lemma required for completeness.

Lemma A.8 (Completeness helper)**.** *Let* Γ *be an environment,* e *an expression,* $\mathsf{fiv}(\theta) = \emptyset$. *Below we also assume that* $dom(\theta)$ *is a subset of the unification variables of the entities it is applied to. We use notation* $\psi \uplus \theta$ *to mean the disjoint union of the two substitutions, and we assume that constraint generation always introduces fresh unification variables, e.g. even disjoint from* θ. *We can then show the following lemmas:*

1. *If* $\theta\Gamma \vdash_{\Downarrow} e : \theta\rho$ *and* $\Gamma \vdash_{\Downarrow} e : \rho \rightsquigarrow C$ *then there exists* ψ *with* $\psi \uplus \theta \models C$.
2. *If* $\theta\Gamma \vdash^{\forall}_{\Downarrow} e : \theta\sigma$ *and* $\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C$ *then there exists a* ψ *with* $\psi \uplus \theta \models C$.
3. *If* $\theta\Gamma \vdash_{\Uparrow} e : \rho_d$ *and* $\Gamma \vdash_{\Uparrow} e : \rho_a \rightsquigarrow C$ *then there exists* ψ *with* $\psi \uplus \theta \models C$ *and* $\psi \uplus \theta(\rho_a) = \rho_d$.
4. *If* $\theta\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \sigma_d$ *and* $\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \sigma_a \rightsquigarrow C$ *then* $\psi \uplus \theta \models C$ *and* $\psi \uplus \theta(\sigma_a) = \sigma_d$.

In all cases, $\mathsf{fiv}(\psi) = \emptyset$.

With these it is possible to show completeness.

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

Figure 9. Extra syntax for inference

B Extensions to the language

As discussed in Section 4.5, one of the salient features of Quick Look is its modularity with respect to other type system features. In this section we describe how some language extensions supported by GHC can be readily integrated in our formalization: **let** bindings, pattern matching, quantified constraints, and GADTs. For the sake of readability, Figure 13 includes all the required syntactic extensions for the forecoming developments.

B.1 Local Bindings

Our description of local bindings, given in Figure 14a, follows Vytiniotis et al. [26] closely. In particular, 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 like in Appendix A.

Figure 14b 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 which type variables to generalize. Another possibility, taken by Hage and Heeren [9], Pottier and Rémy [21], is to extend

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \rightsquigarrow C_{\mathsf{h}} \quad \overline{e} = \mathrm{valargs}(\overline{\pi}) \\ \vdash^{\mathsf{inst}} \sigma[\overline{\pi}] \rightsquigarrow \Delta \;;\; \overline{\phi}, \rho_r \;;\; C_{\mathsf{inst}} \\ \overline{\Gamma \vdash_{\downarrow} e_i : (\Delta, \phi_i) \rightsquigarrow \Theta_i} \\ \Theta_r = \begin{cases} \emptyset & \text{if } \delta = \Uparrow \\ \mathrm{match}(\Delta, \rho_r, \rho, \mathrm{mnv}) & \text{if } \delta = \Downarrow \end{cases} \\ \Theta = (\oplus \Theta_i) \oplus \Theta_r \quad \overline{\phi_i' = \Theta \phi_i} \quad \overline{\forall a. \rho_r' = \Theta \rho_r} \\ \overline{\alpha}, \overline{\beta} \text{ fresh} \quad \theta = [\overline{a} := \overline{\alpha}, (\Delta \setminus \mathrm{dom}(\Theta)) := \overline{\beta}] \\ \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta \phi_i' \rightsquigarrow C_i} \quad \vdash_{\delta} \theta \rho_r' \equiv \rho \rightsquigarrow C_r \end{array}}{\Gamma \vdash_{\delta} h\, \overline{\pi} : \rho \rightsquigarrow C_{\mathsf{h}} \wedge C_{\mathsf{inst}} \wedge \overline{C_i} \wedge C_r} \text{ APPX}$$

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

$$\frac{\beta_a, \beta_r \text{ fresh} \qquad \Gamma, x : \beta_a \vdash_{\Downarrow} e : \beta_r \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.\, e : \alpha \rightsquigarrow C \wedge (\alpha \sim \beta_a \to \beta_r)} \text{ ABS-}\Downarrow\text{-VAR}$$

$$\frac{\Gamma, x : \sigma_a \vdash^{\forall}_{\Downarrow} e : \sigma_r \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.\, e : \sigma_a \to \sigma_r \rightsquigarrow C} \text{ ABS-}\Downarrow\text{-FUN}$$

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

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} x : \sigma \rightsquigarrow \epsilon} \text{ H-VAR}$$

$$\frac{\Gamma \vdash^{\forall}_{\Downarrow} e : \sigma \rightsquigarrow C}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} (e :: \sigma) : \sigma \rightsquigarrow C} \text{ H-ANNOT}$$

$$\frac{\Gamma \vdash_{\Uparrow} e : \rho \rightsquigarrow C}{\Gamma \vdash^{\mathsf{h}}_{\Uparrow} e : \rho \rightsquigarrow C} \text{ H-INFER}$$

$$\boxed{\vdash_{\delta} \rho_1 \equiv \rho_2 \rightsquigarrow C}$$

$$\frac{}{\vdash_{\Uparrow} \rho \equiv \rho \rightsquigarrow \epsilon} \text{ EQ-}\Uparrow$$

$$\frac{}{\vdash_{\Downarrow} \rho_1 \equiv \rho_2 \rightsquigarrow (\rho_1 \sim \rho_2)} \text{ EQ-}\Downarrow$$

Figure 10. Inference algorithm: expressions

$$\boxed{\vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta \;;\; \overline{\phi}, \rho_r \;;\; C}$$

$$\frac{\emptyset \vdash^{i} \sigma[\overline{e}] \rightsquigarrow \Delta \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C}{\vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta \;;\; \overline{\theta \phi}, \rho_r \;;\; C} \text{ INST}$$

$$\boxed{\begin{array}{c}\Delta \vdash^{i} \sigma[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C \\ \text{Invariants: } \mathrm{dom}(\theta) \subseteq \Delta', \mathrm{fuv}(C) \text{ disjoint from } \Delta'\end{array}}$$

$$\frac{}{\Delta \vdash^{i} \rho_r[] \rightsquigarrow \Delta \;;\; \emptyset \;;\; \rho_r \;;\; \epsilon} \text{ IRESULT}$$

$$\frac{\begin{array}{c}\overline{\kappa} \text{ fresh} \quad \rho' = [\overline{a := \kappa}]\rho \\ \Delta, \overline{\kappa} \vdash^{i} \rho'[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C\end{array}}{\Delta \vdash^{i} (\forall \overline{a}.\, \rho)[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C} \text{ IALL}$$

$$\frac{\Delta \vdash^{i} \sigma_2[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C}{\begin{array}{l}\Delta \vdash^{i} (\sigma_1 \to \sigma_2)[e_1\, \overline{e}] \\ \qquad \rightsquigarrow \Delta' \;;\; \theta \;;\; \sigma_1, \overline{\phi}, \rho_r \;;\; C\end{array}} \text{ IARG}$$

$$\frac{\begin{array}{c}\alpha \notin \Delta \quad \beta, \gamma \text{ fresh} \quad C_\alpha = \alpha \sim (\beta \to \gamma) \\ \Delta \vdash^{i} \gamma[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C\end{array}}{\Delta \vdash^{i} \alpha[e_1\, \overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \beta, \overline{\phi}, \rho_r \;;\; C \wedge C_\alpha} \text{ IVARM}$$

$$\frac{\begin{array}{c}\kappa \in \Delta \quad \mu, \upsilon \text{ fresh} \quad \theta_\kappa = [\kappa := (\mu \to \upsilon)] \\ \Delta, \mu, \upsilon \vdash^{i} \upsilon[\overline{e}] \rightsquigarrow \Delta' \;;\; \theta \;;\; \overline{\phi}, \rho_r \;;\; C\end{array}}{\Delta \vdash^{i} \kappa[e_1\, \overline{e}] \rightsquigarrow \Delta' \;;\; \theta \circ \theta_\kappa \;;\; \mu, \overline{\phi}, \rho_r \;;\; C} \text{ IVARD}$$

Figure 11. Inference algorithm: instantiation

the language of constraints with generalization and instantiation, making the solver aware of the order in which these constraints ought to be solved.

B.2 Pattern Matching

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 rules given in Figure 15 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.

B.3 Quantified Constraints

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

Following Vytiniotis et al. [26] we leave the language of constraints open; in the case of GHC this language includes

QL argument $\Gamma \vdash^{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \Theta \; ; \; C$

Invariants: $\mathrm{dom}(\Theta) \subseteq \Delta \subseteq \mathrm{fuv}(\phi)$

$$\frac{\Gamma \vdash^{\mathsf{h}}_{\not\downarrow} h : \sigma \quad \vdash^{\mathsf{inst}} \sigma[\overline{e}] \rightsquigarrow \Delta_\phi \; ; \; \overline{\phi}, \rho_r \; ; \; C_{\mathsf{inst}} \quad \Gamma \vdash^{\not\downarrow} e_i : (\Delta_\phi, \phi_i) \rightsquigarrow \Theta_i \; ; \; C_i \quad \Theta_\phi = \oplus \Theta_i \quad \vdash^{\not\downarrow} (\Delta_\rho, \rho) \sqsubseteq (\Delta_\phi, \Theta_\phi \rho_r) \rightsquigarrow \Theta_\rho \; ; \; C_\rho}{\Gamma \vdash^{\not\downarrow} h\,\overline{e} : (\Delta_\rho, \rho) \rightsquigarrow \Theta_\rho \; ; \; C_{\mathsf{inst}} \wedge \overline{C_i} \wedge C_\rho} \text{ APP-}\not\downarrow$$

$$\frac{e \text{ not an app. or } \phi \text{ not top-level mono.}}{\Gamma \vdash^{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \emptyset \; ; \; \epsilon} \text{ REST-}\not\downarrow$$

QL sub $\vdash^{\not\downarrow} (\Delta_1, \rho_1) \sqsubseteq (\Delta_2, \rho_2) \rightsquigarrow \Theta \; ; \; C$

Inv: $\mathrm{dom}(\Theta) \subseteq \Delta_1 \subseteq \mathrm{fuv}(\rho_1)$, $\Delta_2 \subseteq \mathrm{fuv}(\rho_2)$

$$\frac{\Delta_2 = \emptyset \quad \text{or} \quad \rho_1 \neq \kappa \in \Delta_1 \quad \vdash^{\not\downarrow}_{\Downarrow} (\Delta_1, \rho_1) \sqsubseteq \rho_2 \rightsquigarrow \Theta \; ; \; C}{\vdash^{\not\downarrow} (\Delta_1, \rho_1) \sqsubseteq (\Delta_2, \rho_2) \rightsquigarrow \Theta \; ; \; C} \text{ SUB-}\not\downarrow$$

$$\frac{\kappa \in \Delta_1 \quad \Delta_2 \neq \emptyset}{\vdash^{\not\downarrow} (\Delta_1, \kappa) \sqsubseteq (\Delta_2, \rho_2) \rightsquigarrow \emptyset \; ; \; \epsilon} \text{ SUB-}\not\downarrow\text{-}\Delta$$

QL matching $\vdash^{\not\downarrow}_{\delta} (\Delta, \rho_1) \sqsubseteq \rho_2 \rightsquigarrow \Theta \; ; \; C$

Inv: $\mathrm{dom}(\Theta) \subseteq \Delta \subseteq \mathrm{fuv}(\rho_1)$, $\mathrm{fuv}(\rho_2)$ disjoint from Δ

$$\frac{\Phi = \mathrm{mgu}(\rho_1, \rho_2) \quad \Theta, C = \mathrm{split}(\Delta, \Phi)}{\vdash^{\not\downarrow}_{\Downarrow} (\Delta, \rho_1) \sqsubseteq \rho_2 \rightsquigarrow \Theta \; ; \; C} \text{ U-GD}$$

$$\frac{\delta = \Uparrow \text{ or U-GD fails}}{\vdash^{\not\downarrow}_{\delta} (\Delta, \rho_1) \sqsubseteq \rho_2 \rightsquigarrow \emptyset \; ; \; \epsilon} \text{ U-DEFAULT}$$

$$\begin{array}{l} \mathrm{split}(\Delta, \Phi) = (\Phi_\Delta \uplus \theta,\ \mathsf{C}(\theta \circ \Phi_m)) \\ \quad \text{where} \quad \Phi_\Delta = \Phi|_\Delta \\ \qquad\qquad \Phi_m = \Phi \setminus \Delta \\ \qquad\qquad \overline{\kappa} = \mathrm{fuv}(\mathrm{rng}(\Phi_m)) \cap \Delta \\ \qquad\qquad \theta = [\overline{\kappa := \alpha}] \quad (\alpha \text{ fresh}) \end{array}$$

Figure 12. Inference algorithm: quick look

type class constraints and equalities with type families. Figure 16 shows that the changes required to support quantified constraints are fairly minimal:

- Environments Γ may now also mention *local constraints*. This is a slight departure from Vytiniotis et al. [26], 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.

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & \ldots \\ & & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\ & & \mid & \mathsf{let}\ x :: \sigma = e\ \mathsf{in}\ e \\ & & \mid & \mathsf{case}\ e_0\ \mathsf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} \\ \text{Constraints} & Q & ::= & \epsilon \mid Q \wedge Q \\ & & \mid & \sigma \sim \phi \\ & & \mid & \ldots \text{ extensible} \\ \text{Poly. types} & \sigma, \phi & ::= & \cdots \mid Q \Rightarrow \sigma \\ \text{Constr. sigs} & ksig & ::= & K : \forall \overline{a}\,\overline{b}.\, Q \Rightarrow \overline{\sigma} \to \mathsf{T}\,\overline{a} \\ \text{Environments} & \Gamma & ::= & \cdots \mid \Gamma, ksig \mid \Gamma, Q \end{array}$$

Figure 13. Syntax for extended language

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

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

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

(a) No generalization

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

(b) With generalization

Figure 14. Local bindings

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

$$\frac{\begin{array}{l} \Gamma \vdash_{\Uparrow} e_0 : \mathsf{T}\,\overline{\sigma} \\ \text{for each branch } K_i\ \overline{x_i} \to e_i \text{ do:} \\ \quad K_i : \forall \overline{a}.\, \overline{v_i} \to \mathsf{T}\,\overline{a} \in \Gamma \\ \quad \Gamma, \overline{x_i : [\overline{a := \sigma}]v_i} \vdash_\delta e_i : \rho \end{array}}{\Gamma \vdash_\delta \mathsf{case}\ e_0\ \mathsf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \rho} \text{ CASE}$$

Figure 15. Pattern matching

- 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).

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

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

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

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

$$\boxed{\Delta \vdash^{i} \sigma[\overline{\pi}] \rightsquigarrow \Delta' \;;\; Q \;;\; \theta \;;\; \overline{\phi}, \rho_r}$$

$$\frac{\begin{array}{c}\overline{\pi} = \epsilon \;\text{ or }\; \pi = e\,\overline{\pi'} \\ \overline{\kappa}\ \text{fresh} \quad \rho' = [\overline{a := \kappa}]\rho \\ Q^* = [\overline{a := \kappa}]Q \\ \Delta, \overline{\kappa} \vdash^{i} \rho'[\overline{\pi}] \rightsquigarrow \Delta' \;;\; Q' \;;\; \theta \;;\; \overline{\phi}, \rho_r\end{array}}{\begin{array}{l}\Delta \vdash^{i} (\forall \overline{a}.\, Q \Rightarrow \rho)[\overline{\pi}] \\ \quad \rightsquigarrow \Delta' \;;\; Q^* \wedge Q' \;;\; \theta \;;\; \overline{\phi}, \rho_r\end{array}}\ \textsc{iall}$$

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

Figure 16. Quantified constraints

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

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

Figure 17. GADTs

This judgment can be freely instantiated, as explained in Vytiniotis et al. [26].

B.4 Generalized Algebraic Data Types

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 updated CASE rules in Figure 17 adds those constraints to the environment.

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.