A Quick Look at Impredicativity

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

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

Draft March 2020. We would appreciate any comments or feedback to the email addresses below.

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

1 INTRODUCTION

Parametric polymorphism backed by Damas-Milner type inference was first introduced in ML [14], and has been enormously influential and widely used. But despite its this impact, it has always suffered from an embarrassing shortcoming: *Damas-Milner type inference, and its many variants, cannot instantiate a type variable with a polymorphic type*; in the jargon, the system is *predicative*.

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

Thus motivated, a long succession of papers have tackled the problem of type inference for impredicativity [2, 11, 12, 24, 26, 27]. 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.

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

- We formalise a higher-rank baseline system (Section 4), and give the changes required for Quick Look (Section 5). A key property of Quick Look is that it requires only highly localised changes to such a specification. In particular, no new forms of types are required, and progams can be elaborated into a statically typed intermediate language based on System F. Some other approaches, such as MLF [2], require substantial changes to the intermediate language, but Quick Look does not.

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

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

Fig. 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 6).
- We give a type inference algorithm for Quick Look (Section 7). This algorithm is based on the now-standard approach of first *generating typing constraints* and then *solving them* [21]. As in the case of the declarative specification, no new forms of types or constraints are needed. Section 7 proves its soundness compared with the declarative specification in Section 5.[1] The implementation is in turn based very closely on this algorithm.
  The constraint generation judgements in Section 7 and appendix 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, 25],
- Because Quick Look's impact is so localised, it is simple to implement, even in a production compiler. Concretely, the implementation of Quick Look in GHC, a production compiler for Haskell, affected only 1% of GHC's inference engine.
- To better support impredicativity, we propose to abandon contravariance of the function arrow (Section 5.4). 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 9).

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 B gives the details for a much richer set of features.

We present our work in the concrete setting of (a tiny subset of) Haskell, but there is nothing Haskell-specific about it. Quick Look could readily be used in any other type inference system. We cover the rich related work in Section 10.

2 MOTIVATION

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

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

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

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

As another example, Augustsson describes an application [1] in which it was crucial to have a function *var* of type

$$var :: RValue\ a \to IO\ (\forall lr.LR\ lr \Rightarrow lr\ a)$$

The function *var* is an IO action that returns a polymorphic value. Yet in Haskell today, this is out of reach; instead you have to define a new *named* type, thus:

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

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

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

But in many cases there is no such problem. Consider (*head ids*) where, again, the types are given in Figure 1. Now there is no choice: the only possibility is to instantiate p with $\forall a.a \to a$. Our idea, just as in much previous work [24], is to exploit that special case. Our overall goals are these:

- *First class polymorphism.* We want forall-types to be first class. A function like list $reverse :: \forall a.[a] \to [a]$ should work as uniformly over $[\forall a.a \to a]$ as it does over $[Int]$ and $[Bool]$, and should do so without type annotations. No mainstream deployed language allows that; and not being able to do so is a fundamental failure of abstraction. Using boxed impredicativity is an anti-modular second best.
- *Predictable type inference:* programmers should acquire a robust mental model of what will typecheck, and what will not. Typically they do so through a process of trial and error, but our formalism in Section 5 is specifically designed to enshrine the common-sense idea that when there is clear evidence (through the argument or result type) about how to instantiate a call, type inference should take advantage of it.
- *Minimize type annotations:* "obviously typable" programs should be typeable without annotation. To substantiate this necessarily-qualitative claim we give numerous examples, especially in Figure 11.
- *Conservative extension of Damas-Milner* and its extensions to type classes, higher rank, etc etc. That is, existing programs continue to typecheck (Section 6.1).
- *Can express all of System F*, perhaps with the addition of type annotations (Section 6.1).
- *Localised, in both specification and implementation.* We seek a system that affects only a small part of the specification, and the implementation, of the type system and its infererence algorithm. Modern type systems, such as that of Haskell, OCaml, or Scala, are subtle and complicated; anything that requires pervasive changes is unlikely to be implemented.

3 THE QUICK LOOK

Our new approach works as follows:

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

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

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

Why is $(head\ ids)$ easier? Because the type variable p in *head*'s type appears *guarded*, under the list type constructor; but not so for *single*. Exploiting this guardedness was the key insight of earlier work [24].

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

3.1 "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:

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

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

This same mechanism works when the "expected" type comes from an enclosing call. Suppose $foo :: [\forall a.a \to a] \to Int$, and consider $foo\ (single\ id)$. The context of the call $(single\ id)$ specifies the result type of the call, just as the type signature did before. We need to "push down" the type expected by the context into an expression, but fortunately this ability is already well established in the form of *bidirectional type inference* [18, 19] as Section 4 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 $[\,] :: \forall p.[p]$. Consider the task of instantiating $[\,]$ in the context of a call $foo\ [\,]$. Since *foo* expects an argument of type $[\forall a.a \to a]$, the only way to instantiate $[\,]$ is with $p := \forall a.a \to a$.

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

3.2 Richer Arguments

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

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

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

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

The alert reader will note that 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.

3.3 Uncurried functions

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

3.4 Interim Summary

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.

4 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 5.4. We handle function application in an unusual way, one that will extend nicely for Quick Look, and we add *visible type application* [7].

Type constructors		$\ni$	$\mathsf{F}, \mathsf{G}, \mathsf{T}, \ldots$	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	$::=$	$\kappa \mid \forall a.\sigma \mid \rho$	
Top-level mono. types	ρ	$::=$	$\tau \mid \mathsf{T}\,\overline{\sigma}$	
Fully mono. types	τ	$::=$	$a \mid \mathsf{T}\,\overline{\tau}$	
Typechecking direction	δ	$::=$	$\Uparrow \mid \Downarrow$	Inference and checking respectively
Application heads	h	$::=$	x	Variable
		$\mid$	$e :: \sigma$	Annotation
		$\mid$	e	(not an application)
Arguments	π	$::=$	$\sigma \mid e$	
Terms / expressions	e	$::=$	$h\, \pi_1\ \ldots\ \pi_n$	Application ($n \geqslant 0$)
		$\mid$	$\lambda x.\, e$	Abstraction
Match flag	ω	$::=$	$\mathsf{ma} \mid \mathsf{mnv}$	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, Ψ	$::=$	$[\overline{\kappa := \sigma}]$	$\mathsf{dom}(\theta)$	Domain of θ
				$\mathsf{rng}(\theta)$	Range of θ
				$\Delta_1 \# \Delta_2$	Δ_1 is disjoint from Δ_2

Fig. 2. Syntax

4.1 Syntax

The syntax of our language is given in Figure 2.

Types. The syntax of types is unsurprising. Type constructors T include the function arrow $(\rightarrow)$, although we usually write it infix. So $(\tau_1 \rightarrow \tau_2)$ is syntactic sugar for $((\rightarrow)\ \tau_1\ \tau_2)$. 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 5.5).

Similar to other works on type inference [5, 11, 26] our syntax uses n-ary application. The term $(h\, \pi_1 \ldots \pi_n)$ applies a *head*, h, to a sequence of zero or more arguments $\pi_1 \ldots \pi_n$. The head can be a variable x, an expression with a type annotation $(e :: \sigma)$, or an expression e other than an application. The intuition is that we want to use information from the arguments to inform instantiation of the function's polymorphic variables. In fact, GHC's implementation *already* treats

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

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

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

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

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

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

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

Fig. 3. Base type system: expressions

application as an n-ary operation to improve error messages. Note also that a lone variable x is a valid expression e; it is just an n-ary application with no arguments.

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

4.2 Bidirectional Typing Rules

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

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

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 $\tau \to \rho$. Note that in inference mode the lambda-bound variable must have a monotype. A term like $\lambda x.\, (x\ \mathit{True}, x\ 3)$ is ill-typed in inference mode, because x (being monomorphic) cannot be applied both to a Boolean and an integer. As is conventional, the type τ appears "out of thin air". When constructing a typing

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

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

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

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

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

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

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

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

Fig. 4. Base instantiation

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

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

4.3 Applications, Instantiation, and Subsumption

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

(1) Infer the (polymorphic) type σ of the function h, using $\vdash^{h}_{\Uparrow}$. Usually the function is a variable x, and in that case we simply look up x in the environment Γ (rule H-VAR in Figure 3).
(2) Instantiate h's type σ with fresh *instantiation variables*, $\kappa, \mu, \ldots$, guided by the arguments $\overline{\pi}$ to which it is applied, using the judgment $\vdash^{\mathsf{inst}}$. This judgement returns
   - a type ϕ_i for each of the value arguments, $e_1 \ldots e_n$;
   - the top-level-monomorphic result type ρ_r of the call; and
   - the *instantiation set* Δ of the freshly-instantiated variables.

   We give some examples below.
(3) Conjure up a "magic substitution" θ that maps each of the instantiation variables in Δ to a monotype. 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, shown in Figure 4, has the form

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

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

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

The instantiation judgement tells us that we can treat the type of the head of the application as $\forall\Delta.\,\overline{\phi} \to \rho_r$, and apply that to the value arguments of $\overline{\pi}$, ignoring type arguments which are already embodied in $\overline{\phi}$ and ρ_r. Moreover, the value arguments of $\overline{\pi}$ correspond 1-1 with the argumennt types $\overline{\phi}$.

During instantiation, rule IALL instantiates a leading $\forall$, while rule IARG decomposes a function arrow. We discuss ITYARG in Section 4.4. 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. $f :: Int \to \forall a.[a] \to [a]$. For example,

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

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

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

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

$$\dfrac{\begin{array}{c} \Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma \qquad \overline{e} = \mathsf{valargs}(\overline{\pi}) \qquad \vdash^{\mathsf{inst}} \sigma\,[\overline{\pi}] \leadsto \Delta\ ;\ \overline{\phi}, \rho_r \\ \overline{\Gamma \vdash_{\lightning} e_i : (\Delta, \phi_i) \leadsto \Theta_i} \\ \Theta_r = \mathsf{if}\ \delta = \Uparrow\ \mathsf{then}\ \emptyset\ \mathsf{else}\ \mathsf{match}(\Delta, \rho_r, \rho, \mathsf{mnv}) \\ \Psi = \bigoplus \Theta_i \oplus \Theta_r \qquad \overline{\phi'_i = \Psi\phi_i} \qquad \forall\overline{a}.\rho'_r = \Psi\rho_r \\ \mathsf{dom}(\theta) = \overline{a} \cup \mathsf{fiv}(\mathsf{rng}(\Psi)) \qquad \overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi'_i} \qquad \rho = \theta\rho'_r \end{array}}{\Gamma \vdash_{\delta} h\,\overline{\pi} : \rho}\ \textsc{app}$$

Fig. 5. Quick look impredicativity: the new APP rule

4.4 Visible Type Application

The $\vdash^{\mathsf{inst}}$ judgement also implements visible type application (VTA) [7], a popular extension offered by GHC. The programmer can use VTA to *explicitly* instantiate a function call. For example, if $xs :: [\mathit{Int}]$ we could say either $(\mathit{head}\ xs)$ or, using VTA, $(\mathit{head}\ @\mathit{Int}\ xs)$.

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[2]. The argument is a polytype σ: we allow impredicative instantiation. For example, consider the call $(\mathit{map}\ @(\forall a.a \to a)\ f)$, where we supply one of the two type arguments that map expects (its type is in Figure 1). The instantiation judgement will then look like:

$$\begin{array}{l} \vdash^{\mathsf{inst}} (\forall p\ q.(p \to q) \to [p] \to [q])\,[\,@(\forall a.a \to a), f\,] \\ \qquad\qquad \leadsto\ \{\kappa\}\ ;\ ((\forall a.a \to a) \to \kappa), [\forall a.a \to a],\quad [\kappa] \end{array}$$

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

The attentive reader may note that our typing rules are sloppy about the lexical scoping of type variables (for example in rule GEN), so that they can appear in user-written type signatures or type arguments. Doing this properly is not hard, using the approach of Eisenberg et al. [6], but the plumbing is distracting so we omit it.

5 QUICK LOOK IMPREDICATIVITY

Building on the baseline of Section 4, we are now ready to present Quick Look, the main contribution of this paper.

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

[2] Note that we do not address all the details of the design of Eisenberg et al. [7]. In particular, we do not account for the difference between "specified"‘ and "inferred" quantifiers.

$$\boxed{\begin{array}{c}\text{QL argument} \quad \Gamma \vdash_{\lightning} e : (\Delta, \phi) \leadsto \Theta \\ \text{Invariants: } \mathsf{dom}(\Theta) \subseteq \Delta \subseteq \mathsf{fiv}(\phi)\end{array}}$$

$$\frac{\Gamma \vdash^{\mathsf{h}}_{\lightning} h : \sigma \qquad \vdash^{\mathsf{inst}} \sigma\,[\overline{e}] \leadsto \Delta_\phi \,;\, \overline{\phi}, \rho_r \qquad \overline{\Gamma \vdash_{\lightning} e_i : (\Delta_\phi, \phi_i) \leadsto \Theta_i} \\ \Theta_\phi = \bigoplus \Theta_i \qquad \omega = \text{if } \Delta_\phi = \emptyset \text{ then } \mathsf{ma} \text{ else } \mathsf{mnv}}{\Gamma \vdash_{\lightning} h\,\overline{e} : (\Delta_\rho, \rho) \leadsto \mathsf{match}(\Delta_\rho, \rho, \Theta_\phi \rho_r, \omega)}\ \textsc{app-}\lightning$$

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

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

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

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

$$\begin{array}{rcll} \mathsf{match}(\Delta, \rho_1, \rho_2, \omega) & = & \emptyset & \text{if } \ \omega = \mathsf{mnv},\ \rho_1 = \kappa \in \Delta \\ & = & \mathsf{mch}(\Delta, \rho_1, \rho_2) & \text{otherwise} \end{array}$$

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

$$\begin{array}{rcll} \mathsf{mch}(\Delta, \mathsf{T}\,\overline{\sigma}, \mathsf{T}\,\overline{\phi}) & = & \bigoplus \left(\overline{\mathsf{mch}(\Delta, \sigma, \phi)}\right) & \\ \mathsf{mch}(\Delta, \forall a.\sigma, \forall a.\phi) & = & \Theta & \text{if } a \notin \mathsf{rng}(\Theta) \\ & & \multicolumn{2}{l}{\text{where } \Theta = \mathsf{mch}(\Delta, \sigma, \phi)} \\ \mathsf{mch}(\Delta, \kappa, \phi) & = & [\kappa := \phi] & \text{if } \kappa \in \Delta \\ \mathsf{mch}(\Delta, \sigma, \phi) & = & \emptyset & \text{otherwise} \end{array}$$

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

$$\Theta_1 \oplus \Theta_2 \quad = \quad [\ \kappa := \mathsf{ljoin}(\Theta_1\kappa, \Theta_2\kappa) \mid \kappa \in \mathsf{dom}(\Theta_1) \cup \mathsf{dom}(\Theta_2)\ ]$$

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

$$\begin{array}{rclr} \mathsf{ljoin}(\mathsf{T}\,\overline{\sigma}, \mathsf{T}\,\overline{\phi}) & = & \mathsf{T}\,\overline{\mathsf{ljoin}(\sigma_i, \phi_i)} & (1) \\ \mathsf{ljoin}(\forall a.\sigma, \forall a.\phi) & = & \forall a.\mathsf{ljoin}(\sigma, \phi) & (2) \\ \mathsf{ljoin}(\kappa, \phi) & = & \begin{cases} \phi & \text{if } \phi \text{ contains a } \forall \\ \kappa & \text{otherwise} \end{cases} & (3) \\ \mathsf{ljoin}(\phi, \kappa) & = & \begin{cases} \phi & \text{if } \phi \text{ contains a } \forall \\ \kappa & \text{otherwise} \end{cases} & (4) \\ \mathsf{ljoin}(\sigma, \phi) & = & \sigma \qquad\qquad\qquad\qquad \text{otherwise} & (5) \end{array}$$

Fig. 6. Quick look impredicativity: supporting judgements

A poly-substitution Θ, Ψ 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 Ψ to be empty, meaning "no useful quick-look information".

5.1 A Quick Look at the Argument

The new APP rule is supported by the quick-look judgements in Figure 6. The main judgement takes a quick look at an argument e:

$$\Gamma \vdash_{\lightning} e : (\Delta, \phi) \rightsquigarrow \Theta$$

Here e is the argument, ϕ is the type expected by the function, and Θ is advice (in the form of a substitution for some of the variables in Δ) gained from comparing e with ϕ. The rules of this judgement are given in Figure 6. Rule REST-$\lightning$ is the base case for $\vdash_{\lightning}$. 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 $\vdash_{\lightning}$ do anything at all (Section 3.2).

Looking now at rule APP-$\lightning$, 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-$\lightning$ specialised to a lone variable x:

$$\frac{x : \forall \overline{a}.\rho \in \Gamma \quad \Delta_\phi = \overline{\kappa} \text{ fresh} \quad \rho_r = [\overline{a := \kappa}]\rho \quad \omega = \text{if } \Delta_\phi = \emptyset \text{ then ma else mnv}}{\Gamma \vdash_{\lightning} x : (\Delta, \rho) \rightsquigarrow \mathsf{match}(\Delta, \rho, \rho_r, \omega)}$$

We look up x in Γ, instantiate any top-level quantifiers, and 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), from variables in Δ to polytypes σ. We discuss the first equation in Section 5.2. The second equation delegates to mch, which recursively traverses the two types to find a matching substitution, returning the empty substitution on a mis-match. It uses $\oplus$ to combine poly-substitutions from different parts of the type, just as we do in rule APP to combine poly-substitutions from different arguments of the function.

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 $\Theta = [\kappa := \forall a.a \to a]$.

It is always sound for quick-look matching to return an empty poly-substitution, $\Theta = \emptyset$. That simply means that rule APP must find a mono-substitution θ that makes the program typecheck, just as was the case previously (Figure 3). The quick look step finds whatever polymorphic instantiation information it can; but it never fails.

5.2 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 bails 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 ω = 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 $\textsc{app}$-$\lightning$.

If, on the other hand, there is no instantiation in the argument ($\Delta_\phi = \emptyset$), $\textsc{app}$-$\lightning$ 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 arises in rule $\textsc{app}$ (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 $\textsc{app}$ 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 (*head ids*) :: *Int* $\to$ *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

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

If we instead used $\omega = \mathsf{ma}$ we would compute $\Theta_r = [\kappa := Int \to Int]$, 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.

5.3 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 $\textsc{app}$, 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 allows plenty of leeway in the quick-look part, because once the instantiation is chosen, the arguments will always be typechecked as normal.

Calls as arguments. In rule $\textsc{app}$-$\lightning$, we generalise the rule given in Section 5.1 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 $\textsc{app}$-$\lightning$ sets ω based on whether Δ_ϕ is empty, which neatly checks for instantiation *anywhere* in the argument. For example consider (*single* (f 3)) where $f :: Int \to \forall a.a \to a$, where the instantiation is hidden under an arrow.

Instantiation after quick look. In rule $\textsc{app}$ while ρ_r is top-level monomorphic, $\Theta\rho_r$ may not be. For example, consider (*head 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 $\textsc{app}$ 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. Note that, since θ is a mono-substitution, these $\overline{a}$ are instantiated only with *monotypes*.

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 ljoin to combine the substitutions from independent quick looks. A tricky example is the call $(f\ x\ y)$ where

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

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

$$\Theta_x = [\kappa := (v_b, \sigma_{id})] \quad \Theta_y = [\kappa := (\sigma_{id}, v_c)]$$

where v_b, v_c are the instantiation variables used to instantiate x and y respectively. Joining these leads to the desired final $\Theta = [\kappa := (\sigma_{id}, \sigma_{id})]$. The first component of the tuple is obtained from equation (3) of ljoin, the second from equation (4).

The same holds within a single call of match. Consider the call $(g\ xy)$ where

$$g :: \forall a.(a, a) \to Int \quad xy :: \forall bc.((b, \sigma_{id}), (\sigma_{id}, c))$$

where we want to discover the instantiation $[\kappa := (\sigma_{id}, \sigma_{id})]$.

Note that ljoin never fails; if it encounters a mis-match the program is un-typeable anyway, and ljoin simply returns its first argument (equation (5)). A crucial property of ljoin is that makes Quick Look stable under a mono-substitution θ; that is $\theta(\mathsf{ljoin}(\sigma_1, \sigma_2)) = \mathsf{ljoin}(\theta\sigma_1, \theta\sigma_2)$; see Lemma A.2 in Appendix A. This property is a key part of our soundness proof.

5.4 Co- and Contravariance of Function Types

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

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

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

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

Second, such mismatches are rare (we give data in Section 9), 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. Indeed, a GHC Proposal to simplify the language by adopting an invariant function arrow has been adopted by the community, independently of impredicativity [17]. In Section 9 we quantify the impact of this change in the broader Haskell ecosystem.

5.5 Modularity

Quick Look is like many other works in that it exploits programmer-supplied type annotations to guide type inference (Section 10). But Quick Look's truly distinctive feature is that it is *modular* and *highly localised.*

Highly localised. Through half-closed eyes the changes in Figures 5 and 6 seem substantial. However, 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 [25]. *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 B gives the extra rules for a much larger language; and we have built a full implementation in GHC (Section 8). This implementation is the first working implementation of impredicativity in GHC, despite several attempts over the last decade, each of which became mired in complexity.

6 PROPERTIES OF QUICK LOOK

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

6.1 Expressiveness and Backward Compatibility

Out system is able to type any program typeable in System F, maybe with additional annotations. In order to do so, we define a type-directed translation from System F into our source language, in a very similar fashion to Serrano et al. [24]. Every variable, application, and abstraction is recursively translated, and in addition:

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

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

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

One of our stated goals was to be a conservative extension of Damas-Milner polymorphism.

THEOREM 6.2 (COMPATIBILITY WITH RANK-1 POLYMORPHISM). *Let e be an expression with type τ under λ-calculus with predicative polymorphism, in an environment Γ whose types only have top-level polymorphism. Then $\Gamma \vdash e : \tau$ in the system presented in this paper.*

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

6.2 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 6.3. *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.

6.3 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. For example $app\ f\ x$ should be typeable iff $revapp\ x\ f$ is.

CONJECTURE 6.4 (ARGUMENT PERMUTATION). *Assume environment Γ and expressions $\overline{e_i}$. 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. (Sketch) We give a sketch of the argument, but leave a formal treatment as future work. We need two auxiliary properties:

- First, quick look never disagrees with the main typing judgement on expressions. More precisely, for any Γ, e and ϕ, let $\Delta = \mathsf{fiv}(\phi)$, and let Θ be the quick-look substititution defined by $\Gamma \vdash_{\not\downarrow} e : (\Delta, \phi) \rightsquigarrow \Theta$. Then for any poly-subsitution Ψ such that $\mathrm{dom}(\Psi) = \Delta$ and $\Gamma \vdash^{\forall}_{\Downarrow} e : \Psi(\phi)$, it must be that $\Psi = \Psi_r \cdot \Theta$, for some residual poly-substitution Ψ_r. That is, the quick-look substitution Θ is "on the way" to *any* valid instantiation of the function.
- Second, we observe that if σ_1 and σ_2 are unifiable by a poly-substitution Ψ then $\mathsf{ljoin}(\sigma_1, \sigma_2)$ and $\mathsf{ljoin}(\sigma_2, \sigma_1)$ can only differ in positions inside the two types where one contains an instantiation variable κ_1 and the other an instantiation variable κ_2.

With these two properties in mind, consider an application $(f\ e_1\ e_2)$ that is well-typed by rule APP in Figure 5. Assume we get back Θ_1 from quick-looking at e_1, and Θ_2 from e_2. If Ψ is the master substitution that completes the typing derivation in rule APP, then by the first property above it must be that $\Psi^1_r \cdot \Theta_1 = \Psi = \Psi^2_r \cdot \Theta_2$ for some residual substitutions Ψ^1_r and Ψ^2_r. That in turn means that the ranges of the two substitutions Θ_1 and Θ_2 must be unifiable. Now by the second property above, $\Theta_1 \oplus \Theta_2$ and $\Theta_2 \oplus \Theta_1$ can only differ by some free instantiation variables in their ranges. However these are going to be grounded to monomorphic types by the mono-substitution θ in rule APP, and so do not matter for impredicative instantiations. □

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

Fig. 7. Extra syntax for inference

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 the typing of the argument. For example, suppose that $f :: ((\forall a.a \to a) \to \mathit{Int}) \to \mathit{Bool}$. Then let-abstracting the argument can render the program ill-typed; but it can be fixed by adding a type signature:

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

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 $f :: \forall a.(\mathit{Int} \to a) \to a$. Then we have

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

The trouble here is that Quick Look does not look inside lambda arguments. Again, a type signature makes the program robust to such transformation. 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 5.4, η-expansion is necessary to make a program typecheck. But sometimes the reverse is the case. For example, whereas *app runST argST* is accepted, $\mathit{app}\ (\lambda x\,.\ \mathit{runST}\ x)\ \mathit{argST}$ 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 $\forall s.\mathit{ST}\ s\ v$ must be left as it is or instantiated further.

7 TYPE INFERENCE

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

7.1 Inference algorithm

Typically type inference algorithms work in two stages: *generating* constraints, and then *solving* them, as described in *OutsideIn(X): Modular type inference with local assumptions* [25], 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 B demonstrate.

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\vdash_{\Uparrow} \rho \equiv \rho \rightsquigarrow \epsilon}\ \textsc{Eq-}\Uparrow \qquad \frac{}{\vdash_{\Downarrow} \rho_1 \equiv \rho_2 \rightsquigarrow (\rho_1 \sim \rho_2)}\ \textsc{Eq-}\Downarrow$$

Fig. 8. Inference algorithm: expr. and quick look

Our algorithm generates *constraints* whose syntax is shown in Figure 7. Simple constraints are bags of *equality constraints* $\phi_1 \sim \phi_2$, but we will also need mixed-prefix constraints $\forall\overline{a}.\exists\overline{\alpha}.C$. These forms are not new; they are described in MTILA, and used in GHC's constraint solver. This is a key point of our approach: it requires zero changes to GHC's actual constraint language and solver.

Algorithm: expressions. Figures 8 and 9 present constraint generation for expressions. They closely follow the declarative specification in Figures 3 and 4. For example, the judgement $\Gamma \vdash_\delta e : \rho \rightsquigarrow C$ is very similar to that in Figure 3, but in addition generates constraints C. The big difference is that instead of clairvoyantly selecting monomorphic types τ for λ-abstraction arguments and for other instantiations (e.g. the range of substitution θ in rule APP) the constraint-generation rules create fresh *unification variables*, α, β, γ. Unification variables stand for monomorphic types, and are solved during a subsequent *constraint solving* pass. In contrast, instantiation variables stand for polytypes, and are solved immediately by Quick Look; the constraint solver never sees them. Hence the following invariant:

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

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

Rules ABS-$\Downarrow$-VAR and ABS-$\Uparrow$ 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 8 generates fresh unification variables $\overline{\alpha}$. Second, rule APP in Figure 8 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 9 collects constraints that are generated in rule IVARM. In that case we have a unification variable α that we have to further unify to a function type $\beta \to \gamma$. Note how the constraint C only mentions unification *but no instantiation variables.* Instantiation variables κ are born and eliminated in a single quick look.

Algorithm: quick look, matching and joining. These judgements remain exactly as in Figure 6, since we have carefully designed them to operate only on instantiation variables and ignore unification variables.

7.2 Soundness of Type Inference

We write $\theta \models C$ to mean that a *monomorphic* idempotent substitution θ (from any sort of variables) is a solution to constraint C[3]. Due to Lemma 7.1, a solution θ to a constraint C need only refer to unification variables, but *never* instantiation variables, that must be resolved only through the QL mechanism. The main soundness theorem follows:

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

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 $\mathsf{ljoin}(\phi_1, \phi_2)$ function:

LEMMA 7.3. *If* $\Theta^\star = \bigoplus \overline{\Theta}$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then*

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

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

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

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

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

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

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

$$\frac{\overline{\kappa}\ \text{fresh} \qquad \rho' = [\overline{a := \kappa}]\rho \qquad \Delta, \overline{\kappa} \vdash^{i} \rho'\,[\overline{e}] \rightsquigarrow \Delta' \,;\, \theta \,;\, \overline{\phi}, \rho_r \,;\, C}{\Delta \vdash^{i} (\forall \overline{a}.\, \rho)\,[\overline{e}] \rightsquigarrow \Delta' \,;\, \theta \,;\, \overline{\phi}, \rho_r \,;\, C}\ \textsc{IAll}$$

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

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

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

Fig. 9. Inference algorithm: instantiation

8 IMPLEMENTATION

One of our main claims is that Quick Look can be added, in a modular and non-invasive way, to an existing, production-scale type inference engine. To substantiate the claim, we implemented Quick Look on top of the latest incarnation of GHC (ghc-8.9.0.20191101). 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.[4]

It is remarkable that there is so little to say about the implementation. By way of comparison, attempts to implement Guarded Instantation [24] in GHC floundered in a morass of complexity.

9 THE IMPACT OF AN INVARIANT FUNCTION ARROW

In Section 5.4 we propose to make the function arrow invariant, and to drop deep instantiation and deep skolemisation. This change is independently attractive, and is the subject of a recently adopted GHC Proposal. How much effect does this change have on existing Haskell code?

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

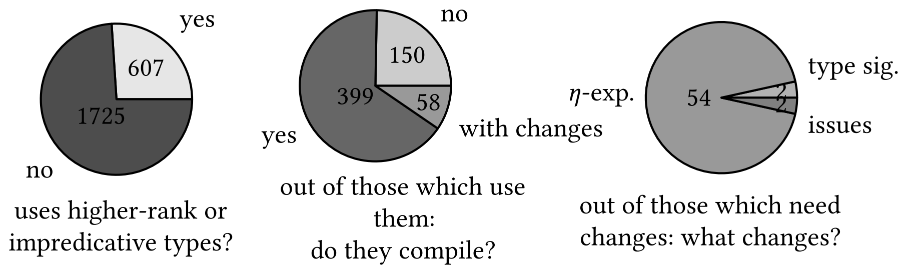

Fig. 10. Summary of the evaluation, using Stackage LTS 13.6

To answer this question we used our implementation to compile a large collection of packages from Hackage; we summarize the results in Figure 10. We started from collection of packages for a recent version of GHC, obtained from Stackage (LTS 13.6), containing a total of 2,332 packages. We then selected only the 607 packages that used the extensions *RankNTypes*, *Rank2Types* or *ImpredicativeTypes*; the other 1,725 packages are certainly unaffected. We then compiled the library sections of each package. Of the 607 packages, 150 fail to compile for reasons unrelated to Quick Look – they depend on external libraries and tools we do not have available; or they do not compile with GHC 8.9 anyway. Of the remaining 457 packages, 399 compiled with no changes whatsoever; two had issues we could not solve, e.g., because of Template Haskell. We leave these two to the authors of these packages.

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

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

10 RELATED WORK

This section explores many different strands of work on first-class polymorphism; another excellent review can be found in [3, §5]. Figure 11 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 [4] re-frame the problem within a more logical setting, and additionally describe extensions to indexed types [5].

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.

		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	$single\ inc \mathbin{+\!\!+} single\ id$	✓	✓	✓	✓	✓	✓
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 GI are not accepted by the vanilla system, but are allowed with some of its extensions [24].

Fig. 11. Comparison of impredicative type systems

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 of Poly-ML [9]. 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 [8] 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 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 [2], but there are several subsequent variants, including HML [12], and FPH [27].

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.

11 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 impredicative instantiation, we expect *type variables in patterns* [6] 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] Lennart Augustsson. 2011. Impredicative polymorphism: a use case. http://augustss.blogspot.com/2011/07/impredicative-polymorphism-use-case-in.html.

[2] 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

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

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

[5] 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

[6] 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

[7] 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

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

[9] 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.

[10] 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).

[11] Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008.* 283–294. https://doi.org/10.1145/1411204.1411245

[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.* 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* (Charleston, South Carolina, USA) *(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* (Tallinn, Estonia) *(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* (Edinburgh, Scotland) *(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] 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

[26] 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

[27] Dimitrios Vytiniotis, Stephanie Weirich, and Simon L. Peyton Jones. 2008. FPH: first-class polymorphism for Haskell. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. 295–306. https://doi.org/10.1145/1411204.1411246

A PROOFS

Section 6 already presents several properties of our constraint generation judgements. One interesting lemma is that if we can infer a type, we can then check it (Theorem 6.3.) We proceed to give the crux of the argument here.

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

Proof. The proof is straightforward induction, with difficulties coming from rule app. Let's assume that $e = h\ \overline{\pi}$. From rule app we'd get:

$$\begin{array}{ll}
\Gamma \vdash^{\mathsf{h}}_{\Uparrow} h : \sigma & (1) \\
\quad \overline{e} = \mathsf{valargs}(\overline{\pi}) & (2) \\
\vdash^{\mathsf{inst}} \sigma\,[\overline{\pi}] \rightsquigarrow \Delta\ ;\ \overline{\phi}, \rho_r & (3) \\
\overline{\Gamma \vdash_{\lightning} e_i : (\Delta, \phi_i) \rightsquigarrow \Theta_i} & (4) \\
\Theta = (\oplus\,\Theta_i) \quad \overline{\phi'_i = \Theta\phi_i} \quad \forall\overline{a}.\rho'_r = \Theta\rho_r & (5) \\
\mathsf{dom}(\theta) = \overline{a} \cup (\Delta \setminus \mathsf{dom}(\Theta)) & (6) \\
\overline{\Gamma \vdash^{\forall}_{\Downarrow} e_i : \theta\phi'_i} & (7) \\
\rho = \theta\rho'_r & (8)
\end{array}$$

Assume now that we'd like to check $\Gamma \vdash_{\Downarrow} e : \theta\rho'_r$. Conditions (1), (2), and (3) directly carry over. However now we need to perform $\mathsf{match}(\Delta, \rho_r, \theta\rho'_r, \mathsf{mnv})$. Let's consider the case first where $\overline{a} = \emptyset$, for simplicity.

In that case: $\theta\rho'_r = \theta\Theta\rho_r$, and we will be invoking the judgement with: $\mathsf{match}(\Delta, \rho_r, \theta\Theta\rho_r, \mathsf{mnv})$. Now, the $dom(\theta)$ contains only variables in the range of Θ (it just monomorphises whatever variables are left over uninstantiated from Θ). Hence it must be the case that matching give a substitution which is equal to $\theta\Theta$ on the free instantiation variables of ρ that are matched to other types. Subsequently, we'd get a $\Theta_{\Downarrow} = (\oplus\,\Theta_i) \oplus (\theta{\cdot}(\oplus\,\Theta_i))$. Joining these two substitutions would create a new substitution that could possibly differ by the original $\theta\Theta$ in that some bindings might not have θ applied to them. Therefore we could always apply $\theta\Theta_{\Downarrow}$ to get a substitution equivalent to $\theta\Theta$. We can The rest follows by (6), (7), (8), above.

The case when $\overline{a} \neq \emptyset$ involves a case analysis. If the original ρ_r was a naked variable mapped by Θ to a forall type or not. If that was *not* the case then a similar analysis as before is applicable.

If however, $\rho_r = \kappa$ then the matching in Figure 6 must *also* return an empty substitution, therefore the inference and checking cases are now indistinguishable. This argument relies crucially on the delicate design of the matching function. □

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

Lemma A.2. *Let* θ *be a monomorphic substitution. If* $\mathsf{ljoin}(\sigma_1, \sigma_2) = \sigma_3$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\mathsf{ljoin}(\theta\sigma_1, \theta\sigma_2) = \theta\sigma_3$.

Proof. By induction on the structure of types σ_1 and σ_2, and case analysis on the ljoin function. □

Lemma A.3. *If* $\mathsf{match}(\Delta, \rho_1, \rho_2, \omega) = \Theta$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\mathsf{match}(\Delta, \theta\rho_1, \theta\rho_2, \omega) = (\theta{\cdot}\Theta)|_{dom(\Theta)}$.

Proof. By induction on the structure of types, appealing to Lemmma A.2. □

Lemma A.4. *If* $\Gamma \vdash^{\mathsf{h}}_{\lightning} h : \sigma$ *and* $\mathsf{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash^{\mathsf{h}}_{\lightning} h : \theta\sigma$.

Proof. Easy case analysis. □

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

Fig. 12. Syntax for extended language

Lemma A.5. *If* $\vdash_\delta \rho_1 \equiv \rho_2 \rightsquigarrow C$ *and* $\theta \models C$ *then* $\theta\rho_1 = \theta\rho_2$.

Proof. Easy case analysis. □

Lemma A.6. *If* $\vdash^{\mathsf{inst}} \sigma\,[\overline{e}] \rightsquigarrow \Delta\ ;\ \overline{\phi}, \rho_r\ ;\ C$ *and* $\theta \models C$ *and* $\mathrm{fiv}(\theta) = \emptyset$ *then:* $\vdash^{\mathsf{inst}} \theta\sigma\,[\overline{e}] \rightsquigarrow \Delta\ ;\ \overline{\theta\phi}, \theta\rho_r$.

Proof. Straightforward induction. The only interesting case is for rule ivarm, where we have to apply rule iarg in the declarative specification. The rest of the cases go through by directly invoking the induction hypothesis or are trivial. □

Lemma A.7. *If* $\Gamma \vdash_{\not{\iota}} e : (\Delta, \phi) \rightsquigarrow \Theta$ *and* $\mathrm{fiv}(\theta) = \emptyset$ *then* $\theta\Gamma \vdash_{\not{\iota}} e : (\Delta, \theta\phi) \rightsquigarrow \theta{\cdot}\Theta|_\Delta$.

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

Lemma A.8 (Soundness). *The following are true (proof by mutual induction on the size of the term):*

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

Theorem 7.2 follows directly from Lemma A.8.

B EXTENSIONS TO THE LANGUAGE

As discussed in Section 5.5, 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 12 includes all the required syntactic extensions for the forecoming developments.

B.1 Local Bindings

Our description of local bindings, given in Figure 13a, follows Vytiniotis et al. [25] closely. In particular, the type of a **let** binding is not generalized unless an explicit annotation is given. This

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

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

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

(a) No generalization

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

(b) With generalization

Fig. 13. Local bindings

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

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

Fig. 14. Pattern matching

design enables the most information to propagate between the definition of a local binding and its use sites when using a constraint-based formulation.

Figure 13b 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 [10], Pottier and Rémy [21], is to extend the language of constraints with generalization and instantiation, making the solver aware of the order in which these constraints ought to be solved.

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 14 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.

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

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

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

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

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

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

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

Fig. 15. Quantified constraints

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

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

Fig. 16. GADTs

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. [25] we leave the language of constraints open; in the case of GHC this language includes type class constraints and equalities with type families. Figure 15 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. [25], in which variable environments and local constraints were kept in separate sets; this change allows us to control better the scope of each type variable.
- Rules GEN and IAPP now have to deal with constraints. In the former case, Q is added to the set of local constraints. In the case of IAPP, the constraints are returned as part of the instantiation judgment.
- Rule APP needs to check that the constraints obtained from instantiation hold for the chosen set of types. We use an auxiliary *constraint entailment* judgment $\Gamma \Vdash Q$ which states that constraints Q hold in the given environment (which may contain local assumptions). This judgment can be freely instantiated, as explained in Vytiniotis et al. [25].

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 16 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.