Visible Type Application

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed

University of Pennsylvania
{eir,sweirich}@cis.upenn.edu
hamidhasan14@gmail.com

Abstract. The Hindley-Milner (HM) type system automatically infers the types at which polymorphic functions are used. In HM, the inferred types are unambiguous, and every expression has a principal type. Type annotations make HM compatible with extensions where complete type inference is impossible, such as higher-rank polymorphism and type-level functions. However, programmers cannot use annotations to explicitly provide type arguments to polymorphic functions, as HM requires type instantiations to be inferred.

We describe an extension to HM that allows visible type application. Our extension requires a novel type inference algorithm, yet its declarative presentation is a simple extension to HM. We prove that our extended system is a conservative extension of HM and admits principal types. We then extend our approach to a higher-rank type system with bidirectional type-checking. We have implemented this system in the Glasgow Haskell Compiler and show how our approach scales in the presence of complex type system features.

1 Introduction

The Hindley-Milner (HM) type system [7, 13, 18] achieves remarkable concision. While allowing a strong typing discipline, a program written in HM need not mention a single type. The brevity of HM comes at a cost, however: HM programs *must* not mention a single type. While this rule has long been relaxed by allowing visible type annotations (and even requiring them for various type system extensions), it remains impossible for languages based on HM, such as OCaml and Haskell, to use *visible type application* when calling a polymorphic function.[1]

This restriction makes sense in the HM type system, where visible type application is unnecessary, as all type instantiations can be determined via unification. Suppose the function *id* has type $\forall\ a.\ a \to a$. If we wished to visibly instantiate the type variable a (in a version of HM extended with type annotations),

[1] Syntax elements appearing in a programmer's source code are often called *explicit*, in contrast to *implicit* terms, which are inferred by the compiler. However, the implicit/explicit distinction also can indicate whether terms are computationally significant [19]. Our work applies only to the inferred vs. specified distinction, so we use *visible* to refer to syntax elements appearing in source code.

Declarative		Syntax-directed		
HM	(§4.1)	C	(§5.1)	from Damas and Milner [8] and Clément et al. [5]
HMV	(§4.2)	V	(§5.2)	HM types with visible type application
B	(§6.2)	SB	(§6.1)	Higher-rank types with visible type application

Fig. 1. The type systems studied in this paper

we could write the expression ($id :: Int \to Int$). This annotation forces the type checker to unify the provided type $Int \to Int$ with the type $a \to a$, concluding that type a should be instantiated with Int.

However, this annotation is a roundabout way of providing information to the type checker. It would be much more direct if programmers could provide type arguments explicitly, writing the expression $id\ @Int$ instead.

Why do we want visible type application? In the Glasgow Haskell Compiler (GHC) – which is based on HM but extends it significantly – there are two main reasons:

First, type instantiation cannot always be determined by unification. Some Haskell features, such as type classes [28] and GHC's type families [3, 4, 11], do not allow the type checker to unambiguously determine type arguments from an annotation. The current workaround for this issue is the $Proxy$ type which clutters implementations and requires careful foresight by library designers. Visible type application improves such code. (See Sect. 2.)

Second, even when type arguments *can* be determined from an annotation, this mechanism is not always friendly to developers. For example, the variable to instantiate could appear multiple times in the type, leading to a long annotation. Partial type signatures help [29], but they do not completely solve the problem. [2]

Although the idea seems straightforward, adding visible type applications to the HM type system requires care, as we describe in Sect. 3. In particular, we observe that we can allow visible type application only at certain types: those with *specified type quantification*, known to the programmer via type annotation. Such types may be instantiated either visibly by the programmer, or when possible, invisibly through inference.

This paper presents a systematic study of the integration of visible type application within the HM typing system. In particular, the contributions of this paper are the four novel type systems (HMV, V, SB, B), summarized in Fig. 1. These systems come in pairs: a declarative version that justifies the compositionality of our extensions and a syntax-directed system that explains the structure of our type inference algorithm.

- System HMV extends the declarative version of the HM type system with a single, straightforward new rule for visible type application. In support of this feature, it also includes two other extensions: scoped type variables and

[2] The extended version of this paper [12] contains an example of this issue.

a distinction between specified and generalized type quantification. The importance of this system is that it demonstrates that visible type application can be added orthogonally to the HM type system, an observation that we found obvious only in hindsight.
- System V is a syntax-directed version of HMV. This type system directly corresponds to a type inference algorithm, called $\mathcal{V}$. Although Algorithm $\mathcal{V}$ works differently than Algorithm $\mathcal{W}$ [8], it retains the ability to calculate principal types. The key insight is that we can *delay* the instantiation of type variables until necessary. We prove that System V is sound and complete with respect to HMV, and that Algorithm $\mathcal{V}$ is sound and complete with respect to System V. These results show the principal types property for HMV.
- System SB is a syntax-directed bidirectional type system with higher-rank types, *i.e.* higher-rank types. In extending GHC with visible type application, we were required to consider the interactions of System V with all of the many type system extensions featured in GHC. Most interactions are orthogonal, as expected from the design of V. However, GHC's extension to support higher-rank types [23] changes its type inference algorithm to be bidirectional. System SB shows that our approach in designing System V straightforwardly extends to a bidirectional system. System SB's role in this paper is twofold: to show how our approach to visible type application meshes well with type system extensions, and to be the basis for our implementation in GHC.
- System B is a novel, simple declarative specification of System SB. We prove that System SB is sound and complete with respect to System B. A similar declarative specification was not present in prior work [23]; this paper shows that an HM-style presentation is possible even in the case of higher-rank systems.

Our visible type application extension is part of GHC 8.0. The extended version [12] describes this implementation and elaborates on interactions between our algorithm and other features of GHC.[3]

2 Why visible type application?

Before we discuss how to extend HM type systems with visible type application, we first elaborate on why we would like this feature in the first place.

When a Haskell library author wishes to give a client the ability to control type variable instantiation, the current workaround is the standard library's *Proxy* type.

```
data Proxy a = Proxy
```

[3] Although the type inference algorithm in this paper uses unification to determine type instantiations, following Damas's Algorithm $\mathcal{W}$ [7], the results in this paper are applicable to GHC's implementation based on constraint-solving. What matters here is how constraints are generated (specified by the syntax-directed versions of type systems) not how they are solved.

However, as we shall see, programming with *Proxy* is noisy and painfully indirect. With built-in visible type application, these examples are streamlined and easier to work with.[4] In the following example and throughout this paper, unadorned code blocks are accepted by GHC 7.10, blocks with a solid gray bar at the left are ill-typed, and blocks with a gray background are accepted only by our implementation of visible type application.

Resolving type class ambiguity Suppose a programmer wished to normalize the representation of expression text by running it through a parser and then pretty printer. The *normalize* function below maps the string `"7 - 1 * 0 + 3 / 3"` to `"((7 - (1 * 0)) + (3 / 3))"`, resolving precedence and making the meaning clear.[5]

```
normalize :: String → String
normalize x = show ((read :: String → Expr) x)
```

However, the designer of this function cannot make it polymorphic in a straightforward way. Adding a polymorphic type signature results in an ambiguous type, which GHC rightly rejects, as it cannot infer the instantiation for *a* at call sites.

```
normalizePoly :: ∀ a. (Show a, Read a) ⇒ String → String
normalizePoly x = show ((read :: String → a) x)
```

Instead, the programmer must add a *Proxy* argument, which is never evaluated, to allow clients of this polymorphic function to specify the parser and pretty-printer to use

```
normalizeProxy :: ∀ a. (Show a, Read a)
               ⇒ Proxy a → String → String
normalizeProxy _ x = show ((read :: String → a) x)

normalizeExpr :: String → String
normalizeExpr = normalizeProxy (Proxy :: Proxy Expr)
```

With visible type application, we can write these two functions more directly:[6]

[4] Visible type application has been a GHC feature request since 2011. See `https://ghc.haskell.org/trac/ghc/ticket/5296`.

[5] This example uses the following functions from the standard library,

```
show :: ∀ a. Show a ⇒ a → String
read :: ∀ a. Read a ⇒ String → a
```

as well as user-defined instances of the *Show* and *Read* classes for the type *Expr*.

[6] Our new extension `TypeApplications` implies the extension `AllowAmbiguousTypes`, which allows our updated *normalize* definition to be accepted.

```
normalize :: ∀ a. (Show a, Read a) ⇒ String → String
normalize x = show (read @a x)
normalizeExpr :: String → String
normalizeExpr = normalize @Expr
```

Although the *show*/*read* ambiguity is somewhat contrived, proxies are indeed useful in more sophisticated APIs. For example, suppose a library designer would like to allow users to choose the representation of an internal data structure to best meet the needs of their application. If the type of that data structure is not included in the input and output types of the API, then a *Proxy* argument is a way to give this control to clients.[7]

Other examples More practical examples of the need for visible type application require a fair amount of build-up to motivate the need for the intricate types involved. We have included two larger examples in the extended version [12]. One builds from recent work on deferring constraints until runtime [2] and the other on translating a dependently typed program from Agda [16] into Haskell.

3 Our approach to visible type application

Visible type application seems like a straightforward extension, but adding this feature – both to GHC and to the HM type system that it is based on – turned out to be more difficult and interesting than we first anticipated. In particular, we encountered two significant questions.

3.1 Just *what* are the type parameters?

The first problem is that it is not always clear what the type parameters to a polymorphic function are!

One aspect of the HM type system is that it permits expressions to be given multiple types. For example, the identity function for pairs,

$$pid\ (x, y) = (x, y)$$

can be assigned any of the following most general types:

(1) $\forall\ a\ b.\quad (a, b) \to (a, b)$
(2) $\forall\ a\ b.\quad (b, a) \to (b, a)$
(3) $\forall\ c\ a\ b.\ (a, b) \to (a, b)$

All of these types are principal; no type above is more general than any other. However, the type of the expression,

[7] See `http://stackoverflow.com/questions/27044209/haskell-why-use-proxy`

Class constraints do not have a fixed ordering in types, and it is possible that a type variable is mentioned *only* in a constraint. Which of the following is preferred?

$\forall\ r\ m\ w\ a.\ (\mathit{MonadReader}\ r\ m, \mathit{MonadWriter}\ w\ m) \Rightarrow a \to m\ a$
$\forall\ w\ m\ r\ a.\ (\mathit{MonadWriter}\ w\ m, \mathit{MonadReader}\ r\ m) \Rightarrow a \to m\ a$

Equality constraints and GADTs can add new quantified variables. Should we prefer the type $\forall\ a.\ a \to a$ or the equivalent type $\forall\ a\ b.\ (a \sim b) \Rightarrow a \to b$?

Type abbreviations mean that quantifying variables as they appear can be ambiguous without also specifying how type abbreviations are used and when they are expanded. Suppose

type $\mathit{Phantom}\ a = \mathit{Int}$
type $\mathit{Swap}\ a\ b = (b, a)$

Should we prefer $\forall\ a\ b.\ \mathit{Swap}\ a\ b \to \mathit{Int}$ or $\forall\ b\ a.\ \mathit{Swap}\ a\ b \to \mathit{Int}$? Similarly, should we prefer $\forall\ a.\ \mathit{Phantom}\ a \to \mathit{Int}$ or $\mathit{Int} \to \mathit{Int}$?

Fig. 2. Why specified polytypes?

$\mathit{pid}\ @\mathit{Int}\ @\mathit{Bool}$

is very different depending on which "equivalent" type is chosen for *pid*:

```
(Int, Bool) → (Int, Bool)   -- pid has type (1)
(Bool, Int) → (Bool, Int)   -- pid has type (2)
∀ b. (Bool, b)  → (Bool, b)     -- pid has type (3)
```

Of course, there are ad hoc mechanisms for resolving this ambiguity. We could try to designate one of the above types (1–3) as the real principal type for *pid*, perhaps by disallowing the quantification of unused variables (ruling out type 3 above) and by enforcing an ordering on how variables are quantified (preferring type 1 over type 2 above). Our goal would be to make sure that each expression has a *unique* principal type, with respect to its quantified type variables. However, in the context of the full Haskell language, this strategy fails. There are just too many ways that types that are not α-equivalent can be considered equivalent by HM. See Fig. 2 for a list of language features that cause difficulties.

In the end, although it may be possible to resolve all of these ambiguities, we prefer not to. That approach leads to a system that is fragile (a new extension could break the requirement that principal types are unique up to α-equivalence), difficult to explain to programmers (who must be able to determine which type is principal) and difficult to reason about.

Our solution: specified polytypes Our system is designed around the following principle:

> *Only user-specified type parameters can be instantiated via explicit type applications.*

In other words, we allow visible type application to instantiate a polytype only when that type is given by a user annotation. This restriction follows in a long line of work requiring user annotations to support advanced type system features [14, 22, 23]. We refer to variables quantified in type annotations as *specified variables*, distinct from compiler-generated quantified variables, which we call *generalized variables*.

There is one nuance to this rule in practice. Haskell allows programming to omit variable quantification, allowing a type signature like

$$const :: a \to b \to a \quad \text{-- NB: no } \forall$$

Are these variables specified? We have decided that they are. There is a very easy rule at work here: just order the variables left-to-right as the user wrote them. We thus consider variables from type signatures to be specified, even when not bound by an explicit $\forall$.

3.2 Is our extension compatible with the rest of the type system?

We do not want to extend just the type inference algorithm that GHC uses. We would also like to extend its *specification*, which is rooted in HM. This way, we will have a concise description (and better understanding) of what programs type check, and a simple way to reason about the properties of the type system.

Our first attempt to add type application to GHC was based on our understanding of Algorithm $\mathcal{W}$, the standard algorithm for HM type inference. This algorithm instantiates polymorphic functions only at occurrences of variables. So, it seems that the only new form we need to allow is a visible type right after variable occurrences:

$$x\ @\tau_1\ ...\ @\tau_n$$

However, this extension is not very robust to code refactoring. For example, it is not closed under substitution. If type application is only allowed at variables, then we cannot substitute for this variable and expect the code to still type check. Therefore our algorithm should allow visible type applications at other expression forms. But where else makes sense?

For example, it seems sensible to allow a type instantiation is after a polymorphic type annotation (such an annotation certainly specifies the type of the expression):

$$(\lambda x \to x :: \forall\ a\ b.\ (a, b) \to (a, b))\ @Int$$

Likewise, we should also allow a visible instantiation after a **let** to enable refactoring:[8]

$$(\textbf{let}\ y = ((\lambda x \to x) :: \forall\ a\ b.\ (a, b) \to (a, b))\ \textbf{in}\ y)\ @Int$$

[8] In fact, the Haskell 2010 Report [15] *defines* type annotations by expanding to a **let**-declaration with a signature.

However, how do we know that we have identified all sites where visible type applications should be allowed? Furthermore, we may have identified them all for core HM, but what happens when we go to the full language of GHC, which includes features that may expose new potential sites?

One way to think about this issue in a principled way is to develop a compositional specification of the type system, which allows type application for *any* expression that can be assigned a polytype. Then, if our algorithm is complete with respect to this specification, we will know that we have allowed type applications in all of the appropriate places. This specification is itself useful in its own right, as we will have a concise description (and better understanding) of what programs type check and a simple way to reason about the properties of the type system.

Once we started thinking of specifications, we found that Algorithm $\mathcal{W}$ could not be matched up with the compositional specification that we wanted. That led us to reconsider our algorithm and develop a new approach to HM type inference.

Our solution: lazy instantiation for specified polytypes Our new type inference algorithm, which we call Algorithm $\mathcal{V}$, is based on the following design principle:

> *Delay instantiation of "specified" type parameters until absolutely necessary.*

Although Algorithm $\mathcal{W}$ instantiates all polytypes immediately, it need not do so. In fact, it is possible to develop a sound and complete alternative implementation of the HM type system that does not do this immediate instantiation. Instead, instantiation is done only on demand, such as when a polymorphic function is applied to arguments. Lazy instantiation has been used in (non-HM) type inference before [10] and may be folklore; however this work contains the first proof that it can be used to implement the HM type system.

In the next section, we give this algorithm a simple specification, presented as a small extension of HM's existing declarative specification. We then continue with a syntax-directed account of the type system, characterizing where lazy instantiations actually must occur during type checking.

4 HM with visible type application

To make our ideas precise, we next review the declarative specification of the HM type system [7, 13, 18] (which we call System HM), and then show how to extend this specification with visible type arguments.

4.1 System HM

The grammar of System HM is shown in Fig. 3. The expression language comprises the Curry-style typed λ-calculus with the addition of numeric literals (of

HM

Metavariables: x, y term variables; a, b, c type variables; n numeric literals

$$
\begin{array}{lll}
e & ::= x \mid \lambda x.\, e \mid e_1\, e_2 & \text{expressions} \\
& \quad \mid n \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \\
\tau & ::= a \mid \tau_1 \to \tau_2 \mid \mathit{Int} & \text{monotypes} \\
\sigma & ::= \forall\{\overline{a}\}.\, \tau & \text{type schemes} \\
\Gamma & ::= \cdot \mid \Gamma, x{:}\sigma & \text{contexts}
\end{array}
$$

HMV

This grammar extends HM:

$$
\begin{array}{lll}
e & ::= \ldots \mid e\,@\tau \mid (\Lambda\overline{a}.e : \upsilon) & \text{expressions} \\
\tau & ::= \ldots & \text{monotypes} \\
\upsilon & ::= \tau \mid \forall a.\, \upsilon & \text{spec. polytypes} \\
\sigma & ::= \upsilon \mid \forall\{a\}.\, \sigma & \text{type schemes} \\
\Gamma & ::= \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, a & \text{contexts}
\end{array}
$$

We write $(e : \upsilon)$ to mean $(\Lambda \cdot .e : \upsilon)$, $\mathit{ftv}(\sigma)$ to be the set of type variables free in σ, and lift ftv to contexts with $\mathit{ftv}(\overline{x{:}\sigma}, \overline{a})$ to be $\overline{a} \cup \bigcup_i \mathit{ftv}(\sigma_i)$.

Fig. 3. Grammars for Systems HM and HMV

type *Int*) and **let**-expressions. Monotypes are as usual, but we diverge from standard notation in type schemes as they quantify over a possibly-empty *set* of type variables. Here, we write these type variables in braces to emphasize that they should be considered order-independent. We sometimes write τ for the type scheme $\forall\{\,\}.\,\tau$ with an empty set of quantified variables and write $\forall\{a\}.\,\forall\{\overline{b}\}.\,\tau$ to mean $\forall\{a, \overline{b}\}.\,\tau$. Here – and throughout this paper – we liberally use the Barendregt convention that bound variables are always distinct from free variables.

The declarative typing rules for System HM appear in Fig. 4. (The figure also includes the definition for our extended system, called System HMV, described in Sect. 4.2.) System HM is not syntax-directed; rules HM_Gen and HM_Sub can apply anywhere.

So that we can better compare this system with others in the paper, we make two changes to the standard HM rules. Neither of these changes are substantial; our version types the same programs as the original.[9] First, in HM_Let, we allow the type of a **let** expression to be a polytype σ, instead of restricting it to be a monotype τ. We discuss this change further in Sect. 5.2. Second, we replace the usual instantiation rule with HM_Sub. This rule allows the type of any expression to be converted to any less general type in one step (as determined by the subsumption relation $\sigma_1 \leq_{\mathsf{hm}} \sigma_2$). Note that in rule HM_InstG the lists of variables $\overline{a}_1$ and $\overline{a}_2$ need not be the same length.

4.2 System HMV: HM with visible types

System HMV is an extension of System HM, adding visible type application. A key detail in its design is its separation of specified type variables from those arising from generalization, as initially explored in Sect. 3.1. Types may be generalized at any time in HMV, quantifying over a variable free in a type but

[9] Both ways of the equivalence proof proceed by induction, liberally using instantiation, generalization, and subsumption to bridge the gap between the two systems.

$\boxed{\Gamma \vdash_{\mathsf{hm}} e : \sigma}$ Typing rules for System HM

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{hm}} x : \sigma}\text{HM\_Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{hm}} e : \tau_2}{\Gamma \vdash_{\mathsf{hm}} \lambda x.\, e : \tau_1 \to \tau_2}\text{HM\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{hm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{hm}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{hm}} e_1\ e_2 : \tau_2}\text{HM\_App} \qquad \frac{}{\Gamma \vdash_{\mathsf{hm}} n : \mathit{Int}}\text{HM\_Int}$$

$$\frac{\Gamma \vdash_{\mathsf{hm}} e_1 : \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{hm}} e_2 : \sigma_2}{\Gamma \vdash_{\mathsf{hm}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma_2}\text{HM\_Let}$$

$$\frac{\Gamma \vdash_{\mathsf{hm}} e : \sigma \qquad a \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{hm}} e : \forall\{a\}.\, \sigma}\text{HM\_Gen} \qquad \frac{\Gamma \vdash_{\mathsf{hm}} e : \sigma_1 \qquad \sigma_1 \leq_{\mathsf{hm}} \sigma_2}{\Gamma \vdash_{\mathsf{hm}} e : \sigma_2}\text{HM\_Sub}$$

$\boxed{\sigma_1 \leq_{\mathsf{hm}} \sigma_2}$ HM subsumption

$$\frac{\tau_1[\overline{\tau}/\overline{a}_1] = \tau_2 \qquad \overline{a}_2 \notin \mathit{ftv}(\forall\{\overline{a}_1\}.\, \tau_1)}{\forall\{\overline{a}_1\}.\, \tau_1 \leq_{\mathsf{hm}} \forall\{\overline{a}_2\}.\, \tau_2}\text{HM\_InstG}$$

$\boxed{\Gamma \vdash_{\mathsf{hmv}} e : \sigma}$ Extra typing rules for System HMV

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash_{\mathsf{hmv}} e : \forall a.\, \upsilon}{\Gamma \vdash_{\mathsf{hmv}} e\ @\tau : \upsilon[\tau/a]}\text{HMV\_TApp} \qquad \frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}, \overline{b}.\, \tau \qquad \Gamma, \overline{a} \vdash_{\mathsf{hmv}} e : \tau \qquad \overline{a}, \overline{b} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{hmv}} (\Lambda \overline{a}.e : \upsilon) : \upsilon}\text{HMV\_Annot}$$

$\boxed{\sigma_1 \leq_{\mathsf{hmv}} \sigma_2}$ HMV subsumption

$$\frac{\tau_1[\overline{\tau}/\overline{b}] = \tau_2}{\forall \overline{a}, \overline{b}.\, \tau_1 \leq_{\mathsf{hmv}} \forall \overline{a}.\, \tau_2}\text{HMV\_InstS} \qquad \frac{\upsilon_1[\overline{\tau}/\overline{a}_1] \leq_{\mathsf{hmv}} \upsilon_2 \qquad \overline{a}_2 \notin \mathit{ftv}(\forall\{\overline{a}_1\}.\, \upsilon_1)}{\forall\{\overline{a}_1\}.\, \upsilon_1 \leq_{\mathsf{hmv}} \forall\{\overline{a}_2\}.\, \upsilon_2}\text{HMV\_InstG}$$

$\boxed{\Gamma \vdash \upsilon}$ Type well-formedness

$$\frac{\mathit{ftv}(\upsilon) \subseteq \Gamma}{\Gamma \vdash \upsilon}\text{Ty\_Scoped}$$

Fig. 4. Typing rules for Systems HM and HMV

not free in the typing context. The type variable generalized in this manner is *not* specified, as the generalization takes place absent any direction from the programmer. By contrast, a type variable mentioned in a type annotation *is* specified, precisely because it is written in the program text.

The grammar of System HMV appears in Fig. 3. The type language is enhanced with a new intermediate form υ that quantifies over an ordered list of type variables. (We sometimes write $\forall a.\, \forall b.\, \tau$ as $\forall a, b.\, \tau$.) This form sits between

$\forall\{a, b\}.\, a \to b \leq_{\mathsf{hmv}} \forall\{a\}.\, a \to a$	Works the same as $\leq_{\mathsf{hm}}$ for type schemes
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \mathit{Int} \to \mathit{Int}$	Can instantiate specified vars
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall a.\, a \to \mathit{Int}$	Can instantiate only a *tail* of the specified vars
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall\{a, b\}.\, a \to b$	Variables can be regeneralized
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall\{b\}.\, \mathit{Int} \to b$	Right-to-left nature of HMV_InstS forces regeneralization
$\forall a, b.\, a \to b \not\leq_{\mathsf{hmv}} \forall b.\, \mathit{Int} \to b$	Known vars are instantiated from the right, never the left
$\forall\{a\}.\, a \to a \not\leq_{\mathsf{hmv}} \forall a.\, a \to a$	Specified quantification is more general than generalized quantification

Fig. 5. Examples of HMV subsumption relation

type schemes and monotypes; σs contain υs, which then contain τs.[10] Thus the full form of a type scheme σ can be written as $\forall\{\overline{a}\}, \overline{b}.\, \tau$, including both a set of generalized variables $\{\overline{a}\}$ and a list of specified variables $\overline{b}$. Note that order never matters for generalized variables (they are in a set) while order does certainly matter for specified variables (the list specifies their order). We say that υ is the metavariable for *specified polytypes*, distinct from *type schemes* σ.

Expressions in HMV include two new forms: $e\,@\tau$ instantiates a specified type variable with a monotype τ, while $(\Lambda\overline{a}.e : \upsilon)$ allows us to annotate an expression with its type, potentially binding scoped type variables if the type is polymorphic. Requiring a type annotation in concert with scoped type variable binding ensures that the order of quantification is specified: the type annotation is a specified polytype υ. We do not allow annotation by type schemes σ: if the user writes the type, all quantified variables are considered specified.

Typing contexts Γ in HMV are enhanced with the ability to store type variables. This feature is used to implement scoped type variables, where the type variables $\overline{a}$, bound in $\Lambda\overline{a}.e$, are available for use in types occurring within e.

Typing rules The type system of HMV includes all of the rules of HM plus the new rules and relation shown at the bottom of Fig. 4. The HMV rules inherited from System HM are modified to recur back to System HMV relations: in effect, replace all hm subscripts with hmv subscripts. Note, in particular, rule HM_Sub; in System HMV, this rule refers to the $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$ relation, described below.

The most important addition to this type system is HMV_TApp, which enables visible type application when the type of the expression is quantified over a specified type variable.

[10] The grammar for System HMV redefines several metavariables. These metavariables then have (slightly) different meanings in different sections of this paper, but disambiguation should be clear from context. In analysis relating systems with different grammars (for example, in Lemma 1), the more restrictive grammar takes precedence.

A type annotation $(\Lambda\overline{a}.e : \upsilon)$, typed with HMV_Annot, allows an expression to be assigned a specified polytype. We require the specified polytype to have the form $\forall\overline{a}, \overline{b}.\tau$; that is, a prefix of the specified polytype's quantified variables must be the type variables scoped in the expression.[11] The inner expression e is then checked at type τ, with the type variables $\overline{a}$ (but not the $\overline{b}$) in scope. Types that appear in expressions (such as in type annotations and explicit type applications) may mention only type variables that are currently in scope.

Of course, in the $\Gamma, \overline{a} \vdash_{\mathsf{hmv}} e : \tau$ premise, the variables $\overline{a}$ and $\overline{b}$ may appear in τ. We call such variables *skolems* and say that *skolemizing* υ yields τ. In effect, these variables form new type constants when type-checking e. When the expression e has type τ, we know that e cannot make any assumptions about the skolems $\overline{a}, \overline{b}$, so we can assign e the type $\forall\overline{a}, \overline{b}.\tau$. This is, in effect, *specified* generalization.

The relation $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$ (Fig. 4) implements subsumption for System HMV. The intuition is that, if $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$, then an expression of type σ_1 can be used wherever one of type σ_2 is expected. For type schemes, the standard notion of σ_1 being a more general type than σ_2 is sufficient. However for specified polytypes, we must be more cautious.

Suppose an expression $x\,@\tau_1\,@\tau_2$ type checks, where x has type $\forall a, b.\upsilon_1$. The subsumption rule means that we can arbitrarily change the type of x to some υ, as long as $\upsilon \leq_{\mathsf{hmv}} \forall a, b.\upsilon_1$. Therefore, υ must be of the form $\forall a, b.\upsilon_2$ so that $x\,@\tau_1\,@\tau_2$ will continue to instantiate a with τ_1 and b with τ_2. Accordingly, we cannot, say, allow subsumption to reorder specified variables.

However, it is safe to allow *some* instantiation of specified variables as part of subsumption, as in rule HMV_InstS. Examine this rule closely: it instantiates variables *from the right*. This odd-looking design choice is critical. Continuing the example above, υ could also be of the form $\forall a, b, c.\upsilon_3$. In this case, the additional specified variable c causes no trouble – it need not be instantiated by a visible application. But we cannot allow instantiation *left-to-right* as that would allow the visible type arguments to skip instantiating a or b.

Further examples illustrating $\leq_{\mathsf{hmv}}$ appear in Fig. 5.

4.3 Properties of System HMV

We wish System HMV to be a conservative extension of System HM. That is, any expression that is well-typed in HM should remain well-typed in HMV, and any expression not well-typed in HM (but written in the HM subset of HMV) should also not be well-typed in HMV.

Lemma 1 (Conservative Extension for HMV). *Suppose Γ and e are both expressible in HM; that is, they do not include any type instantiations, type annotations, scoped type variables, or specified polytypes. Then, $\Gamma \vdash_{\mathsf{hm}} e : \sigma$ if and only if $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$.*

[11] Note that the Barendregt convention allows bound variables to α-vary, so only the number of scoped type variables is important.

This property follows directly from the definition of HMV as an extension of HM. Note, in particular, that no HM typing rule is changed in HMV and that the $\leq_{\mathsf{hmv}}$ relation contains $\leq_{\mathsf{hm}}$; furthermore, the new rules all require constructs not found in HM.

We also wish to know that making generalized variables into specified variables does not disrupt types:

Lemma 2 (Extra knowledge is harmless). *If* $\Gamma, x{:}\forall\{\overline{a}\}.\,\tau \vdash_{\mathsf{hmv}} e : \sigma$*, then* $\Gamma, x{:}\forall\overline{a}.\,\tau \vdash_{\mathsf{hmv}} e : \sigma$*.*

This property follows directly from the context generalization lemma below, noting that $\forall\overline{a}.\,\tau \leq_{\mathsf{hmv}} \forall\{\overline{a}\}.\,\tau$.

Lemma 3 (Context generalization for HMV). *If* $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$ *and* $\Gamma' \leq_{\mathsf{hmv}} \Gamma$*, then* $\Gamma' \vdash_{\mathsf{hmv}} e : \sigma$*.*

This lemma is proved in the extended version [12].

In practical terms, Lemma 2 means that if an expression contains $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2$, and the programmer figures out the type assigned to x (say, $\forall\{\overline{a}\}.\,\tau$) and then includes that type in an annotation (as $\mathbf{let}\, x = (e_1 : \forall\overline{a}.\,\tau)\, \mathbf{in}\, e_2$), the outer expression's type does not then change.

However, note that, by design, context generalization is not as flexible for specified polytypes as it is for type schemes. In other words, suppose the following expression type-checks.

$$\mathbf{let}\ x = ((\lambda y \to y) :: \forall\ a\ b.\ (a, b) \to (a, b))\ \mathbf{in}\ \ldots$$

The programmer cannot then replace the type annotation with the type $\forall\ a.\ a \to a$, because x may be used with visible type applications. This behavior may be surprising, but it follows directly from the fact that $\forall\ a.\ a \to a \not\leq_{\mathsf{hmv}} \forall\ a\ b.\ (a, b) \to (a, b)$.

Finally, we would also like to show that HMV retains the principal types property, defined with respect to the enhanced subsumption relation $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$.

Theorem 4 (Principal types for HMV). *For all terms* e *well-typed in a context* Γ*, there exists a type scheme* σ_{p} *such that* $\Gamma \vdash_{\mathsf{hmv}} e : \sigma_{\mathrm{p}}$ *and, for all* σ *such that* $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$*,* $\sigma_{\mathrm{p}} \leq_{\mathsf{hmv}} \sigma$*.*

Before we can prove this, we first must show how to extend HM's type inference algorithm (Algorithm $\mathcal{W}$ [8]) to include visible type application. Once we do so, we can prove that this new algorithm always computes principal types.

5 Syntax-directed versions of HM and HMV

The type systems in the previous section declare when programs are well-formed, but they are fairly far removed from an algorithm. In particular, the rules HM_Gen and HM_Sub can appear at any point in a typing derivation.

$\boxed{\Gamma \vdash_{\mathsf{c}} e : \tau}$ Typing rules for System C

$$\frac{x{:}\forall\{\overline{a}\}.\,\tau \in \Gamma}{\Gamma \vdash_{\mathsf{c}} x : \tau[\overline{\tau}/\overline{a}]}\mathrm{C\_Var} \qquad \frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{c}} e : \tau_2}{\Gamma \vdash_{\mathsf{c}} \lambda x.\, e : \tau_1 \to \tau_2}\mathrm{C\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{c}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{c}} e_1\ e_2 : \tau_2}\mathrm{C\_App} \qquad \frac{}{\Gamma \vdash_{\mathsf{c}} n : \mathit{Int}}\mathrm{C\_Int}$$

$$\frac{\Gamma \vdash_{\mathsf{c}}^{gen} e_1 : \sigma \qquad \Gamma, x{:}\sigma \vdash_{\mathsf{c}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{c}} \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2}\mathrm{C\_Let}$$

$\boxed{\Gamma \vdash_{\mathsf{c}}^{gen} e : \sigma}$ Generalization for System C

$$\frac{\overline{a} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Gamma) \qquad \Gamma \vdash_{\mathsf{c}} e : \tau}{\Gamma \vdash_{\mathsf{c}}^{gen} e : \forall\{\overline{a}\}.\,\tau}\mathrm{C\_Gen}$$

Fig. 6. Syntax-directed version of the HM type system

5.1 System C

We can explain the HM type system in a more algorithmic manner by using a syntax-directed specification, called System C, in Fig. 6. This version of the type system, derived from Clément et al. [5], clarifies exactly where generalization and instantiation occur during type checking. Notably, instantiation occurs only at the usage of a variable, and generalization occurs only at a **let**-binding. These rules are syntax-directed because the conclusion of each rule in the main judgment $\Gamma \vdash_{\mathsf{c}} e : \tau$ is syntactically distinct. Thus, from the shape of an expression, we can determine the shape of its typing derivation.

However, the judgment $\Gamma \vdash_{\mathsf{c}} e : \tau$ is still not quite an algorithm: it makes nondeterministic guesses. For example, in the rule C_Abs, the type τ_1 is guessed; there is no indication in the expression what the choice for τ_1 should be. The advantage of studying a syntax-directed system such as System C is that doing so separates concerns: System C fixes the structure of the typing derivation (and of any implementation) while leaving monotype-guessing as a separate problem. Algorithm $\mathcal{W}$ deduces the monotypes via unification, but a constraint-based approach [25, 27] would also work.

5.2 System V: Syntax-directed visible types

Just as System C is a syntax-directed version of HM, we can also define System V, a syntax-directed version of HMV (Fig. 7). However, although we could define HMV by a small addition to HM (two new rules, plus subsumption), the difference between System C and System V is more significant.

Like System C, System V uses multiple judgments to restrict where generalization and instantiation can occur. In particular, the system allows an expres-

$\boxed{\Gamma \vdash_{\mathsf{v}} e : \tau}$ Monotype checking for System V

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{v}} e : \tau_2}{\Gamma \vdash_{\mathsf{v}} \lambda x.\, e : \tau_1 \to \tau_2}\mathrm{V\_Abs} \qquad \frac{\Gamma \vdash_{\mathsf{v}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{v}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{v}} e_1\ e_2 : \tau_2}\mathrm{V\_App}$$

$$\frac{}{\Gamma \vdash_{\mathsf{v}} n : \mathit{Int}}\mathrm{V\_Int} \qquad \frac{\Gamma \vdash^{*}_{\mathsf{v}} e : \forall \overline{a}.\, \tau \quad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{v}} e : \tau[\overline{\tau}/\overline{a}]}\mathrm{V\_InstS}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon}$ Specified polytype checking for System V

$$\frac{x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{v}} x : \upsilon[\overline{\tau}/\overline{a}]}\mathrm{V\_Var} \qquad \frac{\Gamma \vdash^{gen}_{\mathsf{v}} e_1 : \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{v}} e_2 : \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{v}} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \upsilon_2}\mathrm{V\_Let}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{*}_{\mathsf{v}} e : \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{v}} e\ @\tau : \upsilon[\tau/a]}\mathrm{V\_TApp} \qquad \frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}, \overline{b}.\, \tau \qquad \Gamma, \overline{a} \vdash_{\mathsf{v}} e : \tau \qquad \overline{a}, \overline{b} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash^{*}_{\mathsf{v}} (\Lambda \overline{a}.e : \upsilon) : \upsilon}\mathrm{V\_Annot}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} e : \tau \quad \text{no other rule matches}}{\Gamma \vdash^{*}_{\mathsf{v}} e : \tau}\mathrm{V\_Mono}$$

$\boxed{\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma}$ Generalization for System V

$$\frac{\overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma) \qquad \Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon}{\Gamma \vdash^{gen}_{\mathsf{v}} e : \forall\{\overline{a}\}.\, \upsilon}\mathrm{V\_Gen}$$

Fig. 7. Typing rules for System V

sion to have a type scheme only as a result of generalization (using the judgment $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma$). Generalization is, once again, available only in **let**-expressions.

However, the main difference that enables visible type annotation is the separation of the main typing judgment into two: $\Gamma \vdash_{\mathsf{v}} e : \tau$ and $\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon$. The key idea is that, sometimes, we need to be lazy about instantiating specified type variables so that the programmer has a chance to add a visible instantiation. Therefore, the system splits the rules into a judgment $\vdash_{\mathsf{v}}$ that requires e to have a monotype, and those in $\vdash^{*}_{\mathsf{v}}$ that can retain specified quantification.

The first set of rules in Fig. 7, as in System C, infers a monotype for the expression. The premises of the rule V_Abs uses this same judgment, for example, to require that the body of an abstraction have a monotype. All expressions can be assigned a monotype; if the first three rules do not apply, the last rule V_InstS infers a polytype instead, then instantiates it to yield a monotype. Because implicit instantiation happens all at once in this rule, we do not need to worry about instantiating specified variables out of order, as we did in System HMV.

The second set of rules (the $\vdash^{*}_{\mathsf{v}}$ judgment) allows e to be assigned a specified polytype. Note that the premise of rule V_TApp uses this judgment.

Rule V_Var is like rule C_Var: both look up a variable in the environment and instantiate its generalized quantified variables. The difference is that C_Var's types can contain *only* generalized variables; System V's types can have specified variables after the generalized ones. Yet we instantiate only the generalized ones in the V_Var rule, lazily preserving the specified ones.

Rule V_Let is likewise similar to C_Let. The only difference is that the result type is not restricted to be a monotype. By putting V_Let in the $\vdash^{*}_{\mathsf{v}}$ judgment and returning a specified polytype, we allow the following judgment to hold:

$$\cdot \vdash_{\mathsf{v}} (\mathbf{let}\, x = (\lambda y.\, y : \forall a.\, a \to a)\, \mathbf{in}\, x)\, @\mathit{Int} : \mathit{Int} \to \mathit{Int}$$

The expression above would be ill-typed in a system that restricted the result of a **let**-expression to be a monotype. It is for this reason that we altered System HM to include a polytype in its HM_Let rule, for consistency with HMV.

Rule V_Annot is identical to rule HMV_Annot. It uses the $\vdash_{\mathsf{v}}$ judgment in its premise to force instantiation of all quantified type variables before regeneralizing to the specified polytype υ. In this way, the V_Annot rule is effectively able to reorder specified variables. Here, reordering is acceptable, precisely because it is user-directed.

Finally, if an expression form cannot yield a specified polytype, rule V_Mono delegates to $\vdash_{\mathsf{v}}$ to find a monotype for the expression.

5.3 Relating System V to System HMV

Systems HMV and V are equivalent; they type check the same set of expressions. We prove this correspondence using the following two theorems.

Theorem 5 (Soundness of V against HMV).

1. *If* $\Gamma \vdash_{\mathsf{v}} e : \tau$*, then* $\Gamma \vdash_{\mathsf{hmv}} e : \tau$*.*
2. *If* $\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon$*, then* $\Gamma \vdash_{\mathsf{hmv}} e : \upsilon$*.*
3. *If* $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma$*, then* $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$*.*

Theorem 6 (Completeness of V against HMV). *If* $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$*, then there exists* σ' *such that* $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma'$ *where* $\sigma' \leq_{\mathsf{hmv}} \sigma$*.*

The proofs of these theorems appear in the extended version [12].

Having established the equivalence of System V with System HMV, we can note that Lemma 2 ("Extra knowledge is harmless") carries over from HMV to V. This property is quite interesting in the context of System V. It says that a typing context where all type variables are specified admits all the same expressions as one where some type variables are generalized. In System V, however, specified and generalized variables are instantiated via different mechanisms, so this is a powerful theorem indeed.

It is mechanical to go from the statement of System V in Fig. 7 to an algorithm. In the extended version [12], we define Algorithm $\mathcal{V}$ which implements

System V, analogous to Algorithm $\mathcal{W}$ which implements System C. We then prove that Algorithm $\mathcal{V}$ is sound and complete with respect to System V and that Algorithm $\mathcal{V}$ finds principal types. Linking the pieces together gives us the proof of the principal types property for System HMV (Theorem 4). Furthermore, Algorithm $\mathcal{V}$ is guaranteed to terminate, yielding this theorem:

Theorem 7. *Type-checking System V is decidable.*

6 Higher-rank type systems

We now extend the design of System HMV to include *higher-rank polymorphism* [17]. This extension allows function parameters to be used at multiple types. Incorporating this extension is actually quite straightforward. We include this extension to show that our framework for visible type application is indeed easy to extend – the syntax-directed system we study in this section is essentially a merge of System V and the bidirectional system from our previous work [23]. This system is also the basis for our implementation in GHC.

As an example, the following function does not type check in the vanilla Hindley-Milner type system, assuming *id* has type $\forall\ a.\ a \to a$.

$$\textbf{let}\ \mathit{foo} = \lambda f \to (f\ 3, f\ \mathit{True})\ \textbf{in}\ \mathit{foo}\ \mathit{id}$$

Yet, with the `RankNTypes` language extension and the following type annotation, GHC is happy to accept

$$\begin{array}{l}\textbf{let}\ \mathit{foo} :: (\forall\ a.\ a \to a) \to (\mathit{Int}, \mathit{Bool}) \\ \quad \mathit{foo} = \lambda f \to (f\ 3, f\ \mathit{True}) \\ \textbf{in}\ \mathit{foo}\ \mathit{id}\end{array}$$

Visible type application means that higher-rank arguments can also be explicitly instantiated. For example, we can instantiate lambda-bound identifiers:

$$\begin{array}{l}\textbf{let}\ \mathit{foo} :: (\forall\ a.\ a \to a) \to (\mathit{Int} \to \mathit{Int}, \mathit{Bool}) \\ \quad \mathit{foo} = \lambda f \to (f\ @\mathit{Int}, f\ \mathit{True}) \\ \textbf{in}\ \mathit{foo}\ \mathit{id}\end{array}$$

Higher-rank types also mean that visible instantiations can occur after other arguments are passed to a function. For example, consider this alternative type for the *pair* function:

$$\begin{array}{l}\mathit{pair} :: \forall\ a.\ a \to \forall\ b.\ b \to (a, b) \\ \mathit{pair} = \lambda x\ y \to (x, y)\end{array}$$

If *pair* has this type, we can instantiate b after providing the first component for the pair, thus:

```
bar = pair 'x' @Bool
  -- bar inferred to have type Bool → (Char, Bool)
```

In the rest of this section, we provide the technical details of these language features and discuss their interactions. In contrast to the presentation above, we present the syntax-directed higher-rank system first. We do so for two reasons: understanding a bidirectional system requires thinking about syntax, and thus the syntax-directed system seems easier to understand; and we view the declarative system as an expression of properties – or a set a metatheorems – about the higher-rank type system.

6.1 System SB: Syntax-directed Bidirectional Type Checking

Figures 8 and 9 show System SB, the higher-rank, bidirectional analogue of System V, supporting predicative higher-rank polymorphism and visible type application.

This system shares the same expression language of Systems HMV and V, retaining visible type application and type annotations. However, types in System SB may have non-prenex quantification. The body of a specified polytype υ is now a *phi-type* ϕ: a type that has no top-level quantification but may have quantification to the left or to the right of arrows. Note also that these inner quantified types are υs, not σs. In other words, non-prenex quantification is over only *specified* variables, never generalized ones. As we will see, inner quantified types are introduced only by user annotation, and thus there is no way the system could produce an inner type scheme, even if the syntactic restriction were not in place.

The grammar also defines *rho-types* ρ, which also have no top-level quantification, but do allow inner quantification to the *left* of arrows. We convert specified polytypes (which may quantify to the right of arrows) to corresponding rho-types by means of the *prenex* operation, which appears in Fig. 9.

System SB is defined by five mutually recursive judgments: $\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi$, $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon$, and $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma$ are synthesis judgments, producing the type as an output; $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ and $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$ are checking judgments, requiring the type as an input.

Type synthesis The synthesis judgments are very similar to the judgments from System V, ignoring direction arrows. The differences stem from the non-prenex quantification allowed in SB. The level of similarity is unsurprising, as the previous systems essentially all work only in synthesis mode; they derive a type given an expression. The novelty of a bidirectional system is its ability to propagate information about specified polytypes toward the leaves of an expression.

Type checking Rule SB_DAbs is what makes the system higher-rank. The checking judgment $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ pushes in a rho-type, with no top-level quantification. Thus, SB_DAbs can recognize an arrow type $\upsilon_1 \to \rho_2$. Propagating this type into an expression $\lambda x.\, e$, SB_DAbs uses the type υ_1 as x's type when

The grammar for SB extends that for System HMV (Fig. 3):

$e ::= \ldots$	expressions	$\rho ::= \tau \mid \upsilon_1 \to \rho_2$	rho-types
$\tau ::= \ldots$	monotypes	$\phi ::= \tau \mid \upsilon_1 \to \upsilon_2$	phi-types
$\Gamma ::= \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, a$	contexts	$\upsilon ::= \phi \mid \forall a.\, \upsilon$	specified polytypes
		$\sigma ::= \upsilon \mid \forall \{a\}.\, \sigma$	type schemes

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi}$ Synthesis of types without top-level quantifiers

$$\frac{\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \to \upsilon}\mathrm{SB\_Abs} \qquad \frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall \overline{a}.\, \phi \quad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi[\overline{\tau}/\overline{a}]}\mathrm{SB\_InstS}$$

$$\frac{}{\Gamma \vdash_{\mathsf{sb}} n \Rightarrow \mathit{Int}}\mathrm{SB\_Int}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}$ Synthesis of specified polytypes

$$\frac{x{:}\forall \{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \upsilon[\overline{\tau}/\overline{a}]}\mathrm{SB\_Var} \qquad \frac{\Gamma \vdash_{\mathsf{sb}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash^{*}_{\mathsf{sb}} e_1\ e_2 \Rightarrow \upsilon_2}\mathrm{SB\_App}$$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} e\ @\tau \Rightarrow \upsilon[\tau/a]}\mathrm{SB\_TApp} \qquad \frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}, \overline{b}.\, \phi \quad \Gamma, \overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi \qquad \overline{a}, \overline{b} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash^{*}_{\mathsf{sb}} (\Lambda \overline{a}.e : \upsilon) \Rightarrow \upsilon}\mathrm{SB\_Annot}$$

$$\frac{\Gamma \vdash^{\mathit{gen}}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{sb}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \upsilon_2}\mathrm{SB\_Let} \qquad \frac{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi \quad \text{no other rule matches}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \phi}\mathrm{SB\_Phi}$$

$\boxed{\Gamma \vdash^{\mathit{gen}}_{\mathsf{sb}} e \Rightarrow \sigma}$ Synthesis with generalization

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon \qquad \overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)}{\Gamma \vdash^{\mathit{gen}}_{\mathsf{sb}} e \Rightarrow \forall \{\overline{a}\}.\, \upsilon}\mathrm{SB\_Gen}$$

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho}$ Checking against types without top-level quantifiers

$$\frac{\Gamma \vdash^{\mathit{gen}}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{sb}} e_2 \Leftarrow \rho_2}{\Gamma \vdash_{\mathsf{sb}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftarrow \rho_2}\mathrm{SB\_DLet}$$

$$\frac{\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \rho_2}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \to \rho_2}\mathrm{SB\_DAbs} \qquad \frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{dsk}} \rho_2 \quad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho_2}\mathrm{SB\_Infer}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}$ Checking against specified polytypes

$$\frac{\mathit{prenex}(\upsilon) = \forall \overline{a}.\, \rho \quad \overline{a} \notin \mathit{ftv}(\Gamma) \qquad \Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}\mathrm{SB\_DeepSkol}$$

Fig. 8. Syntax-directed bidirectional type system

$\boxed{\sigma_1 \leq_\mathsf{b} \sigma_2}$ Higher-rank instantiation

$$\frac{}{\tau \leq_\mathsf{b} \tau}\text{B\_Refl} \qquad \frac{\upsilon_3 \leq_\mathsf{b} \upsilon_1 \qquad \upsilon_2 \leq_\mathsf{b} \upsilon_4}{\upsilon_1 \to \upsilon_2 \leq_\mathsf{b} \upsilon_3 \to \upsilon_4}\text{B\_Fun}$$

$$\frac{\phi_1[\overline{\tau}/\overline{b}] \leq_\mathsf{b} \phi_2}{\forall \overline{a}, \overline{b}.\, \phi_1 \leq_\mathsf{b} \forall \overline{a}.\, \phi_2}\text{B\_InstS} \qquad \frac{\upsilon_1[\overline{\tau}/\overline{a}] \leq_\mathsf{b} \upsilon_2 \qquad \overline{b} \notin \mathit{ftv}(\forall \{\overline{a}\}.\, \upsilon_1)}{\forall \{\overline{a}\}.\, \upsilon_1 \leq_\mathsf{b} \forall \{\overline{b}\}.\, \upsilon_2}\text{B\_InstG}$$

$\boxed{\phi_1 \leq^*_\mathsf{dsk} \rho_2}$ Subsumption, after deep skolemization

$$\frac{}{\tau \leq^*_\mathsf{dsk} \tau}\text{DSK\_Refl} \qquad \frac{\upsilon_3 \leq_\mathsf{dsk} \upsilon_1 \qquad \upsilon_2 \leq_\mathsf{dsk} \rho_4}{\upsilon_1 \to \upsilon_2 \leq^*_\mathsf{dsk} \upsilon_3 \to \rho_4}\text{DSK\_Fun}$$

$\boxed{\sigma_1 \leq_\mathsf{dsk} \upsilon_2}$ Deep skolemization

$$\frac{\begin{array}{l}\mathit{prenex}(\upsilon_2) = \forall \overline{c}.\, \rho_2 \\ \phi_1[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq^*_\mathsf{dsk} \rho_2\end{array}}{\forall \{\overline{a}\}, \overline{b}.\, \phi_1 \leq_\mathsf{dsk} \upsilon_2}\text{DSK\_Inst}$$

Define $\mathit{prenex}(\upsilon) = \forall \overline{a}.\, \rho$ as follows:

$$\begin{array}{ll} \mathit{prenex}(\forall \overline{a}.\, \tau) & = \forall \overline{a}.\, \tau \\ \mathit{prenex}(\forall \overline{a}.\, \upsilon_1 \to \upsilon_2) & = \forall \overline{a}, \overline{b}.\, \upsilon_1 \to \rho_2 \\ \quad \text{where } \forall \overline{b}.\, \rho_2 = \mathit{prenex}(\upsilon_2) & \end{array}$$

Examples:

$$\begin{array}{rcll} \forall a.\, a \to \forall b.\, b \to b & \leq_\mathsf{b} & \mathit{Int} \to \mathit{Bool} \to \mathit{Bool} & \text{Can instantiate non-top-level vars} \\ \forall a.\, a \to \forall b.\, b \to b & \leq_\mathsf{b} & \mathit{Int} \to \forall b.\, b \to b & \text{Not all vars must be instantiated} \\ \forall a.\, a \to \forall b.\, b \to b & \leq_\mathsf{b} & \forall a.\, a \to \mathit{Bool} \to \mathit{Bool} & \text{Can skip a top-level quantifier} \\ (\mathit{Int} \to \mathit{Int}) \to \mathit{Bool} & \leq_\mathsf{b} & (\forall a.\, a \to a) \to \mathit{Bool} & \text{Contravariant instantiation} \\ \mathit{Int} \to \forall a, b.\, a \to b & \not\leq_\mathsf{b} & \mathit{Int} \to \forall b.\, \mathit{Bool} \to b & \text{Spec. vars are inst'd from the right} \\ \mathit{Int} \to \forall a.\, a \to a & \not\leq_\mathsf{b} & \forall a.\, \mathit{Int} \to a \to a & \text{Cannot move } \forall \text{ for spec. vars} \\ \mathit{Int} \to \forall a, b.\, a \to b & \leq_\mathsf{dsk} & \mathit{Int} \to \forall b.\, \mathit{Bool} \to b & \leq_\mathsf{dsk} \text{ can inst. spec. vars in any order} \\ \mathit{Int} \to \forall a.\, a \to a & \leq_\mathsf{dsk} & \forall a.\, \mathit{Int} \to a \to a & \text{Spec. } \forall \text{ can move with } \leq_\mathsf{dsk} \\ (\mathit{Int} \to \forall b.\, b \to b) \to \mathit{Int} & \leq_\mathsf{dsk} & (\forall a, b.\, a \to b \to b) \to \mathit{Int} & \text{Contravariant out-of-order inst.} \\ \forall \{a\}.\, a \to a & \leq_\mathsf{dsk} & \forall a.\, a \to a & \text{Same handling of spec. and gen. vars} \end{array}$$

Fig. 9. Higher-rank subsumption relations

checking e. This is the only place in system SB where a lambda-term can abstract over a variable with a polymorphic type. Note that the synthesis rule SB_Abs uses a monotype for the type of x.[12]

[12] Higher-rank systems can also include an "annotated abstraction" form, $\lambda x{:}\upsilon.\, e$. This form allows higher-rank types to be synthesized for lambda expressions as well as checked. However, this form is straightforward to add but is not part of GHC, which

Rule SB_Infer mediates between the checking and synthesis judgments. When no checking rule applies, we synthesize a type and then check it according to the $\leq_{\mathsf{dsk}}$ deep skolemization relation, taken directly from previous work and shown in Fig. 9. For brevity, we do not explain the details of this relation here, instead referring readers to Peyton Jones et al. [23, Sect. 4.6] for much deeper discussion. However, we note that there is a design choice to be made here; we could have also used Odersky–Läufer's slightly less expressive higher-rank subsumption relation [21] instead. We present the system with deep skolemization for backwards compatibility with GHC. See the extended version [12] for a discussion of this alternative.

The entry point into the type checking judgments is through the $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$ judgment. This judgment has just one rule, SB_DeepSkol. The rule skolemizes all type variables appearing at the top-level and to the right of arrows. Skolemizing here is necessary to expose a rho-type to the $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ judgment, so that rule SB_DAbs can fire.[13] The reason this rule requires deep skolemization instead of top-level skolemization is subtle, but this choice is not due to visible type application or lazy instantiation; the same choice is made in prior work [23, rule gen2 of Fig. 8]. We refer readers to the extended version [12] for the details.

6.2 System B: Declarative specification

Figure 10 shows the typing rules of System B, a declarative system that accepts the same programs as System SB. This declarative type system itself is a novel contribution of this work. (The systems presented in related work [10, 21, 23] are more similar to SB than to B.)

Although System B is *bidirectional*, it is also *declarative*. In particular, the use of generalization (B_Gen), subsumption (B_Sub), skolemization (B_Skol), and mode switching (B_Infer), can happen arbitrarily in a typing derivation. Understanding what expressions are well-typed does not require knowing precisely when these operations take place.

The subsumption rule (B_Sub) in the synthesis judgment corresponds to HMV_Sub from HMV. However, the novel subsumption relation $\leq_{\mathsf{b}}$ used by this rule, shown at the top of Fig. 9, is *different* from the $\leq_{\mathsf{dsk}}$ deep skolemization relation used in System SB. This $\sigma_1 \leq_{\mathsf{b}} \sigma_2$ judgment extends the action of $\leq_{\mathsf{hmv}}$ to higher-rank types: in particular, it allows subsumption for generalized type variables (which can be quantified only at the top level) and instantiation (only) for specified type variables. We could say that this judgment enables *inner instantiation* because instantiations are not restricted to top level. See also the examples at the bottom of Fig. 9.

In contrast, rule B_Infer (in the checking judgment) uses the stronger of the two subsumption relations $\leq_{\mathsf{dsk}}$. This rule appears at precisely the spot in

uses patterns (beyond the scope of this paper) to bind variables in abstractions. Therefore we omit the annotated abstraction form from our formalism.

[13] Our choice to skolemize before SB_DLet is arbitrary, as SB_DLet does not interact with the propagated type.

$\boxed{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma}$ Synthesis rules for System B

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \sigma}\text{B\_Var} \qquad \frac{\Gamma, x{:}\tau \vdash_{\mathsf{b}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Rightarrow \tau \to \upsilon}\text{B\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash_{\mathsf{b}} e_1\ e_2 \Rightarrow \upsilon_2}\text{B\_App} \qquad \frac{}{\Gamma \vdash_{\mathsf{b}} n \Rightarrow \mathit{Int}}\text{B\_Int}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Rightarrow \sigma}{\Gamma \vdash_{\mathsf{b}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow \sigma}\text{B\_Let}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma \qquad a \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall\{a\}.\, \sigma}\text{B\_Gen} \qquad \frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{b}} \sigma_2}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_2}\text{B\_Sub}$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau \\ \Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall a.\, \upsilon\end{array}}{\Gamma \vdash_{\mathsf{b}} e\ @\tau \Rightarrow \upsilon[\tau/a]}\text{B\_TApp} \qquad \frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}, \overline{b}.\, \phi \qquad \Gamma, \overline{a} \vdash_{\mathsf{b}} e \Leftarrow \phi \qquad \overline{a}, \overline{b} \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} (\Lambda \overline{a}.e : \upsilon) \Rightarrow \upsilon}\text{B\_Annot}$$

$\boxed{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon}$ Checking rules for System B

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2}\text{B\_DAbs} \qquad \frac{\begin{array}{l}\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \\ \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon\end{array}}{\Gamma \vdash_{\mathsf{b}} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftarrow \upsilon}\text{B\_DLet}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon \qquad a \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \forall a.\, \upsilon}\text{B\_Skol} \qquad \frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{dsk}} \upsilon_2}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}\text{B\_Infer}$$

Fig. 10. System B

the derivation where a specified type from synthesis mode meets the specified type from checking mode. The relation $\leq_{\mathsf{dsk}}$ subsumes $\leq_{\mathsf{b}}$; that is, $\sigma_1 \leq_{\mathsf{b}} \upsilon_2$ implies $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$.

Properties of System B and SB We can show that Systems SB and B admit the same expressions.

Lemma 8 (Soundness of System SB).

1. *If* $\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi$ *then* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \phi$.
2. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon$ *then* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \upsilon$.
3. *If* $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma$ *then* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$.
4. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$ *then* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$.
5. *If* $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ *then* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \rho$.

Lemma 9 (Completeness of System SB).

1. *If* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$ *then* $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$ *where* $\sigma' \leq_{\mathsf{b}} \sigma$.
2. *If* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$ *then* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$.

What is the role of System B? In our experience, programmers tend to prefer the syntax-directed presentation of the system because that version is more algorithmic. As a result, it can be easier to understand why a program type checks (or doesn't) by reasoning about System SB.

However, the fact that System B is sound and complete with respect to System SB provides properties that we can use to reason about SB. The main difference between the two is that System B divides subsumption into two different relations. The weaker $\leq_{\mathsf{b}}$ can be used at any time during synthesis, but it can only instantiate (but not generalize) specified variables. The stronger $\leq_{\mathsf{dsk}}$ is used at only the check/synthesis boundary but can generalize and reorder specified variables.

The connection between the two systems tells us that B_Sub is *admissible* for SB. As a result, when refactoring code, we need not worry about precisely where a type is instantiated, as we see here that instantiation need not be determined syntactically.

Likewise, the proof also shows that System B (and System SB) is flexible with respect to the instantiation relation $\leq_{\mathsf{b}}$ in the context. As in System HMV, this result implies that making generalized variables into specified variables does not disrupt types.

Lemma 10 (Context Generalization). *Suppose* $\Gamma' \leq_{\mathsf{b}} \Gamma$.

1. *If* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$ *then there exists* $\sigma' \leq_{\mathsf{b}} \sigma$ *such that* $\Gamma' \vdash_{\mathsf{b}} e \Rightarrow \sigma'$.
2. *If* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$ *and* $\upsilon \leq_{\mathsf{b}} \upsilon'$ *then* $\Gamma' \vdash_{\mathsf{b}} e \Leftarrow \upsilon'$.

Proofs of these properties appear in the extended version [12].

6.3 Integrating visible type application with GHC

System SB is the direct inspiration for the type-checking algorithm used in our version of GHC enhanced with visible type application. It is remarkably straightforward to implement the system described here within GHC; accounting for the behavior around imported functions (Sect. 3.1) was the hardest part. The other interactions (the difference between this paper's scoped type variables and GHC's, how specified type variables work with type classes, etc.) are generally uninteresting; see the extended version [12] for further comments.

One pleasing synergy between visible type application and GHC concerns GHC's recent *partial type signature* feature [29]. This feature allows wildcards, written with an underscore, to appear in types; GHC infers the correct replacement for the wildcard. These work well in visible type applications, allowing the user to write @_ as a visible type argument where GHC can infer the argument. For example, if f has type $\forall\ a\ b.\ a \to b \to (a, b)$, then we can write f @_ @[*Int*] *True* [] to let GHC infer that a should be *Bool* but to visibly instantiate b to be [*Int*]. Getting partial type signatures to work in the new context of visible type applications required nothing more than hooking up the pieces.

7 Related work and Conclusions

Explicit type arguments The F# language [26] permits explicit type arguments in function applications and method invocations. These explicit arguments, typically mandatory, are used to resolve ambiguity in type-dependent operations. However, the properties of this feature have not been studied.

Implicit arguments in dependently-typed languages Languages such as Coq [6], Agda [20], Idris [1] and Twelf [24] are not based on the HM type system, so their designs differ from Systems HMV and B. However, they do support invisible arguments. In these languages, an invisible argument is not necessarily a type; it could be any argument that can be inferred by the type checker.

Coq, Agda, and Idris require all quantification, including that for invisible arguments, to be specified by the user. These languages do not support generalization, i.e., automatically determining that an expression should quantify over an invisible argument (in addition to any visible ones). They differ in how they specify the visibility of arguments, yet all of them provide the ability to override an invisibility specification and provide such arguments visibly. These languages also provide a facility for *named* invisible arguments, allowing users to specify argument values by name instead of by position. This choice means that α-equivalent types are no longer fully interchangeable. Though we have not studied this possibility deeply, we conjecture that formally specifying a named-argument system would encounter many of the same subtleties (in particular, requiring two different subsumption relations in the metatheory) that we encountered with positional arguments.

Twelf, on the other hand, supports invisible arguments via generalization and visible arguments via specification. Although it is easy to convert between the two versions, there is no way to visibly provide an invisible argument as we have done. Instead, the user must rely on type annotations to control instantiations.

Specified vs. generalized variables Dreyer and Blume's work on specifying ML's type system and inference algorithm in the presence of modules [9] introduces a separation of (what we call) specified and generalized variables. Their work focuses on the type parameters to ML functors, finding inconsistencies between the ML language specification and implementations. They conclude that the ML specification as written is hard to implement and propose a new one. Their work includes a type system that allows functors to have invisible arguments alongside their visible ones. This specification is easier to implement, as they demonstrate.

Their work has similarities to ours in the separation of classes of variables and the need to alter the specification to make type inference reasonable. Interestingly, they come from the opposite direction from ours, adding invisible arguments in a place where arguments previously were all visible. However, despite these surface similarities, we have not found a deeper connection between our work and theirs.

Predicative, higher-rank type systems As we have already indicated, Systems B and SB are directly inspired by GHC's design for higher-rank types [23]. However, in this work we have redesigned the algorithm to use lazy instantiation and have made a distinction between specified polytypes and generalized polytypes. Furthermore, we have pushed the design further, providing a declarative specification for the type system.

Our work is also closely related to recent work on using a bidirectional type system for higher-rank polymorphism by Dunfield and Krishnaswami [10], called DK below. The closest relationship is between their declarative system (Fig. 4 in their paper) and our System SB (Fig. 8). The most significant difference is that the DK system never generalizes. All polymorphic types in their system are specified; functions must have a type annotation to be polymorphic. Consequently, DK uses a different algorithm for type checking than the one proposed in this work. Nevertheless, it defers instantiations of specified polymorphism, like our algorithm.

Our relation $\leq_{\mathsf{dsk}}$, which requires two specified polytypes, is similar to the DK subsumption relation. The DK version is slightly weaker as it does not use deep skolemization; but that difference is not important in this context. Another minor difference is that System SB uses the $\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi$ judgment to syntactically guide instantiation whereas the the DK system uses a separate application judgment form. System B – and the metatheory of System SB – also includes implicit subsumption $\leq_{\mathsf{b}}$, which does not have an analogue in the DK system. A more extended comparison with the DK system appears in the extended version [12].

Conclusion This work extends the HM type system with visible type application, while maintaining important properties of that system that make it useful for functional programmers. Our extension is fully backwards compatible with previous versions of GHC. It retains the principal types property, leading to robustness during refactoring. At the same time, our new systems come with simple, compositional specifications.

While we have incorporated visible type application with all existing features of GHC, we do not plan to stop there. We hope that our mix of specified polytypes and type schemes will become a basis for additional type system extensions, such as impredicative types, type-level lambdas, and dependent types.

Acknowledgments Thanks to Simon Peyton Jones, Dimitrios Vytiniotis, Iavor Diatchki, Adam Gundry, Conor McBride, Neel Krishnaswami, and Didier Rémy for helpful discussion and feedback.

Bibliography

[1] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.*, 23, 2013.

[2] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *International Conference on Functional Programming*, ICFP '15. ACM, 2015.

[3] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. Associated type synonyms. In *International Conference on Functional Programming*, ICFP '05. ACM, 2005.

[4] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[5] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: Mini-ML. In *Conference on LISP and Functional Programming*, LFP '86. ACM, 1986.

[6] Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[7] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[8] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, POPL '82. ACM, 1982.

[9] Derek Dreyer and Matthias Blume. Principal type schemes for modular programs. In *Proceedings of the 16th European Conference on Programming*, ESOP'07. Springer-Verlag, 2007.

[10] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[11] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[12] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. Visible type application (extended version), 2015. URL `http://www.seas.upenn.edu/~sweirich/papers/type-app-extended.pdf`.

[13] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

[14] Didier Le Botlan and Didier Rémy. ML^{F}: Raising ML to the power of System F. In *International Conference on Functional Programming*. ACM, 2003.

[15] Simon Marlow (editor). Haskell 2010 language report, 2010.

[16] Conor McBride. Agda-curious? Keynote, ICFP'12, 2012.

[17] Nancy McCracken. The typechecking of programs with implicit type structure. In Gilles Kahn, David B. MacQueen, and Gordon Plotkin, editors,

Semantics of Data Types, volume 173 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1984.

[18] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, 1978.

[19] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer Berlin Heidelberg, 2001.

[20] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[21] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Symposium on Principles of Programming Languages*, POPL '96. ACM, 1996.

[22] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, ICFP '06. ACM, 2006.

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

[24] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, number 1632 in LNAI, pages 202–206, Trento, Italy, July 1999. Springer-Verlag.

[25] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference, pages 387–489. The MIT Press, 2005.

[26] Don Syme. *The F# 2.0 Language Specification*. Microsoft Research and the Microsoft Developer Division, 2012. Accessed from `http://fsharp.org/specs/language-spec/2.0/FSharpSpec-2.0-April-2012.pdf`.

[27] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5), September 2011.

[28] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[29] Thomas Winant, Dominique Devriese, Frank Piessens, and Tom Schrijvers. Partial type signatures for haskell. In *Practical Aspects of Declarative Languages*, volume 8324, pages 17–32. Springer International Publishing, January 2014.