Visible Type Application (Extended version)

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Stephanie Weirich
University of Pennsylvania
sweirich@cis.upenn.edu

Hamidhasan G. Ahmed
University of Pennsylvania
hamidhasan14@gmail.com

Abstract

The Hindley-Milner (HM) type system allows programmers to write polymorphic functions and automatically infers the types at which those functions are used. In this type system, the inferred types are always unambiguous, and every expression has a principal type. However, type inference is sometimes unwieldy or impossible – especially in the presence of type system extensions such as type classes and non-injective type-level functions. In these scenarios, programmers cannot provide type arguments explicitly, as HM requires such types to be invisible.

In this paper, we describe an extension to HM that allows for visible type application. Our extension is based on a novel type inference algorithm, yet a declarative presentation of the system can be specified via 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, 11, 17] 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 systems 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), we could write this expression

$$(\mathit{id} :: \mathit{Int} \to \mathit{Int})$$

This annotation forces the type checker to unify the provided type $\mathit{Int} \to \mathit{Int}$ 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 directly, writing the expression

$$\mathit{id}\ @\mathit{Int}$$

instead.

So why do we want visible type application? Considering a language like Haskell – as implemented by the Glasgow Haskell Compiler (GHC) – which is based on HM but extends it significantly, we find two main benefits.

Type instantiation cannot always be determined by unification. Some Haskell features, such as type classes [28] and GHC's type families [3, 4, 10], 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 Section 2.)

It is sometimes painful to determine instantiations via type annotations. Even when type arguments *can* be determined from an annotation, this mechanism is still 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 don't completely solve the problem. Section 2 also contains an example of this issue.

Although the idea seems straightforward, adding visible type applications to the HM type system requires care, as we describe in Section 3. In particular, we observe that we can allow visible type application only at certain types: those with *specified type quantification*. These types are known to the programmer via type annotation. Such types may be instantiated visibly. Their instantiations may also be inferred as usual, should the programmer omit type applications.

Declarative	**Syntax-directed**	
HM (§4.1)	C (§5.1)	from [8, 17] and [5]
HMV (§4.2)	V (§5.2)	HM types with visible type application
B (§6.1)	SB (§6.2)	Higher-rank types with visible type application

Figure 1. The type systems studied in this paper

[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 is sometimes used to indicate whether terms are computationally significant [18]. Our work to applies only to the inferred vs. programmer-specified distinction, so we use *visible* to refer to syntax elements appearing in source code.

This paper presents a systematic study of the integration of visible type application within the HM typing discipline. In particular, we study this new feature in the context of four novel type systems, summarized in Figure 1.

Section 4 presents System HMV, a conservative extension of the declarative HM type system. HMV makes a distinction between specified and generalized type quantification and with support for visible type application. This extended system retains HM's simplicity and compositionality, making programs with visible type applications easy to reason about.

Section 5 develops System V, a novel syntax-directed type system with visible type application that faithfully implements the specification of the previous section. This type system directly corresponds to a type inference algorithm, called $\mathcal{V}$. We show that 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 polymorphic 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 for System V. These results show the principal types property for HMV.

Our goal with this work is to extend the Glasgow Haskell Compiler with visible type application. Doing so requires considering interactions with the many type system extensions featured in that context. Most interactions are orthogonal, but our work has led us to reconsider the treatment of *higher-rank polymorphism* in GHC [23]. As with visible types, that feature is based on reasoning about user-specified polymorphism.

Therefore, Section 6 extends our ideas to a bidirectional system with higher-rank types and, for full expressiveness, scoped type variables. This section includes a syntax-directed set of rules, called System SB, that adapts the design principles of System V to GHC's current algorithm for higher-rank polymorphism. The section also includes a novel, simple, declarative specification of this bidirectional type system, called System B. We prove that System SB is sound and complete with respect to System B.

System SB forms the basis for our implementation in the Glasgow Haskell Compiler.[2] Section 7 describes this implementation and elaborates on interactions between our algorithm and other features of GHC.

Finally, Section 8 discusses related work. Additionally, Appendix G presents a variant of Systems B and SB designed for comparison with the higher-rank type system of Dunfield and Krishnaswami [9].

The Appendices of this paper contain extended examples and detailed proofs of the properties studied about each of the systems.

However, 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. The next section briefly describes two situations in Haskell where visible type applications would benefit programmers.

2. Examples of visible type application

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
```

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.[3] In these examples 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.

2.1 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.[4]

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

However, the designer of this function can't make it polymorphic in a straightforward way. Adding a polymorphic type signature results in an ambiguous type, which GHC rightly rejects.

```
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[5]

```
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 in this case, proxies are indeed useful in more sophisticated APIs. For example, suppose a library design would like to allow users of the library 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 flexibility to clients.[6]

2.2 Dependently-typed programming

Our next example is an excerpt of a longer example of dependently-typed programming in GHC. Space constraints prevent us from

[2] Our implementation is available from `https://github.com/goldfirere/ghc`, at the `popl-2016` tag.

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

[4] These examples use the following functions from the standard library

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

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

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

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

fully explaining the code here: more details about this example are available in Appendix A. This code is inspired by examples from McBride's ICFP 2012 keynote [15].

GHC supports dependently-typed programming through two main features: type-level computation and indexed types. For the former, *type families* [3, 4, 10] and *data type promotion* [30] allow programmers to write functions using datatypes, such as booleans and lists, at the type level. For the latter, *generalized algebraic datatypes* (GADTs) [22] allow type arguments to be non-uniform. For example, the equality GADT, defined below, is inhabited by *Refl* only when its two arguments are equal.

```
data a :~: b where
  Refl :: a :~: a
```

We can use these features in the following function that asserts an equality fact about type-level computation: We can commute a type family *If* with list cons (:). Note that *fact* takes two *Proxy* arguments; the type variables *t* and *f* appear only in arguments to the *If* type family and thus cannot be solved via unification.

```
type family If cond t f where
  If 'True  t f = t
  If 'False t f = f

fact :: ∀ t f s b. Sing b → Proxy t → Proxy f →
  ((If b t f) ': s) :~: (If b (t ': s) (f ': s))
```

This *fact* is needed in the code below, which comes from a simple compiler from a boolean expression language to a stack machine. We use this fact when compiling conditional expressions, as shown below. (Again, more details are in the appendix.) Not only do we need to provide proxies when calling *fact*, but we must also provide a (long) type annotation for its return type. Because the result of *fact* is used as the scrutinee of a GADT pattern-match, GHC cannot use unification to resolve the type variable *s* in this type. Instead, the only way to supply *s* is through this annotation.

```
compile (SCond (se0 :: Sing e0)
  (se1 :: Sing e1) (se2 :: Sing e2)) =
  case (fact (sEval se0)
        (Proxy :: Proxy (Eval e1))
        (Proxy :: Proxy (Eval e2)) ::
     ((If (Eval e0) (Eval e1) (Eval e2)) ': s) :~:
        (If (Eval e0) ((Eval e1) ': s) ((Eval e2) ': s))) of
     Refl → compile se0 ++
       IFPOP (compile se1) (compile se2) ::: Nil
```

This situation could be slightly improved by adding a *Proxy* for *s* to *fact*. However, *s* appears outside of *If* in the type of *fact*, so the programmer may not be aware of the issue when writing *fact*'s type.

In the presence of visible type application, we avoid the proxies altogether

```
fact :: ∀ t f s b. Sing b →
  ((If b t f) ': s) :~: (If b (t ': s) (f ': s))
```

and provide the type arguments directly, without annotation

```
compile (SCond se0 (se1 :: Sing e1) (se2 :: Sing e2)) =
  case fact @(Eval e1) @(Eval e2) @s (sEval se0) of
     Refl → compile se0 ++
       IFPOP (compile se1) (compile se2) ::: Nil
```

Summary In these cases, although *Proxy* solves the problem, the mechanism clutters code and requires library authors to design their functions to take *Proxy* arguments. Furthermore, once a library author has specified that a function should take a *Proxy* argument, then it must always be called with a proxy.

In contrast, visible type application requires little planning from library designers, can be used with less clutter, and need not be used at all in situations where unification can already determine the type argument.

These examples are not the only ones that we have seen. Haskell programmers make frequent use of *Proxy*. Appendix A contains an additional longer example of the benefit of visible type application.

Why specified polytypes?

It may seem possible to characterize how GHC quantifies type variables, in an attempt to define some sort of "canonical quantification" as a part of the type inference process. We could then prefer one version of the principal type of an expression over another, allowing us to predictably visibly instantiate type variables. However, various type system features mean that such a characterization would be terribly complicated. In particular:

Class constraints don't 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?

```
∀ r m w a. (MonadReader r m
           , MonadWriter w m) ⇒ a → m a
∀ w m r a. (MonadWriter w m
           , MonadReader r m) ⇒ a → 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 in order in the term can be ambiguous without also specifying how type abbreviations are used and when they are expanded. Suppose

```
type Phantom a = Int
type Swap a b = (b, a)
```

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

Type families also disturb variable ordering. Suppose

```
type family Swap (a :: ⋆) :: ⋆
type instance Swap (a, b) = (b, a)
```

Should we prefer $\forall\ a\ b.\ Swap\ (a, b) \to Int$ or $\forall\ b\ a.\ Swap\ (a, b) \to Int$? This issue is harder to solve than the difficulty with vanilla type synonyms, as type families may or may not be able to reduce.

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 problems when trying to extend the HM type system with visible type application.

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 assigned any number of isomorphic types. For example, the identity function for pairs

$$\mathit{pid}\ (x, y) = (x, y)$$

can be assigned any of the following types

$$\begin{array}{l} (1)\ \forall\ a\ b.\quad (a, b) \to (a, b) \\ (2)\ \forall\ a\ b.\quad (b, a) \to (b, a) \\ (3)\ \forall\ c\ a\ b.\ (a, b) \to (a, b) \end{array}$$

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

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

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

$$\begin{array}{lll} & (\mathit{Int}, \mathit{Bool}) \to (\mathit{Int}, \mathit{Bool}) & \text{-- pid has type (1)} \\ & (\mathit{Bool}, \mathit{Int}) \to (\mathit{Bool}, \mathit{Int}) & \text{-- pid has type (2)} \\ \forall\ b.\ (\mathit{Bool}, b) & \to (\mathit{Bool}, b) & \text{-- pid has type (3)} \end{array}$$

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 id, perhaps by disallowing the quantification of unused variables (ruling out type 3 above) or 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 the box on the preceding page for a summary.)

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 Therefore, our system is designed around the following principle:

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

In other words, we allow visible type application to instantiate a polytype only when both of the following are true:

1. The polytype is already fixed: constraint solving will give us no more information about the type.
2. The programmer may reasonably know what the type is.

In practice, these guidelines mean that visible type application is available only on types that are given by an annotation. These restrictions follow in a long line of work requiring more user annotations to support more advanced type system features [13, 22, 23]. See Section 7 for discussion on how our implementation in GHC works with our design principle.

3.2 What is the specification of the type system?

We don't 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 can't 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?

One place that seems sensible to allow a type instantiation is after a polymorphic type annotation (such an annotation certainly specifies the type of the expression)

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

Likewise, if we refactor this term as below, we should also allow a visible instantiation after a **let**[7]

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

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 we develop an algorithm that is complete with respect to this specification, we will know that we have allowed type applications in all of the appropriate places.

Our solution: lazy instantiation for specified polytypes This reasoning, inspired by thinking about how to extend the declarative specification of the HM type system, has lead us to develop a *novel* algorithm for type inference. This 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.

In the next section, we give this algorithm a simple specification, presented as a small extension of HM's existing declarative specification. We then make the details of our algorithm precise by giving 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, 17] (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 Figure 2. The expression language comprises the Curry-style typed λ-calculus with the ad-

[7] In fact, the Haskell 2010 Report [14] *defines* type annotations by expanding to a **let**-declaration with a signature.

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

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

Figure 2. Grammar for System HM

dition of numeric literals (of type *Int*) and **let**-expressions. Monotypes are standard, but we quantify over a possibly-empty *set* of type variables in type schemes. Here, we diverge from standard notation and 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 Figure 3. (This figure also includes rules for our extended system, called System HMV, described in Section 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 small changes to the standard HM rules. Neither of these changes are substantial; our version types the same programs as the original. First, 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 Section 5.2.) Second, we replace the usual instantiation rule with HM_SUB. This rule allows the type of any expression to be to 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 a_1 and 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 Section 3.1. Types may be generalized at any time in HMV, quantifying over a variable free in a type but 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.

4.2.1 Grammar

The grammar of System HMV appears in Figure 4. The type language is enhanced with a new intermediate form υ that quantifies over an ordered list of type variables. This form sits between type schemes and monotypes; σs contain υs, which then contain τs.[8] Thus the full form of a type scheme σ is $\forall\{\overline{a}\}, \overline{b}.\,\tau$, 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

[8] 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.

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

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{hm}} x : \sigma}\ \text{HM\_VAR}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{hm}} e : \tau_2}{\Gamma \vdash_{\mathsf{hm}} \lambda x.\, e : \tau_1 \to \tau_2}\ \text{HM\_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{hm}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{hm}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{hm}} e_1\, e_2 : \tau_2}\ \text{HM\_APP}$$

$$\frac{}{\Gamma \vdash_{\mathsf{hm}} n : \mathit{Int}}\ \text{HM\_INT}$$

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

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

$$\frac{\Gamma \vdash_{\mathsf{hm}} e : \sigma_1 \qquad \sigma_1 \leq_{\mathsf{hm}} \sigma_2}{\Gamma \vdash_{\mathsf{hm}} e : \sigma_2}\ \text{HM\_SUB}$$

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

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

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

$$\frac{\tau \text{ closed} \qquad \Gamma \vdash_{\mathsf{hmv}} e : \forall a.\,\upsilon}{\Gamma \vdash_{\mathsf{hmv}} e\,@\tau : \upsilon[\tau/a]}\ \text{HMV\_TAPP}$$

$$\frac{\upsilon \text{ closed} \qquad \upsilon = \forall\overline{a}.\,\tau \qquad \Gamma \vdash_{\mathsf{hmv}} e : \tau}{\Gamma \vdash_{\mathsf{hmv}} (e : \upsilon) : \upsilon}\ \text{HMV\_ANNOT}$$

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

$$\frac{\tau_1[\overline{\tau}/\overline{b}] = \tau_2}{\forall\overline{a}, \overline{b}.\,\tau_1 \leq_{\mathsf{hmv}} \forall\overline{a}.\,\tau_2}\ \text{HMV\_INSTS}$$

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

Figure 3. Typing rules for Systems HM and HMV

The grammar for HMV extends that for System HM (Figure 2):

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

Figure 4. Grammar for System HMV

$\forall\{a,b\}.\, a \to b \leq_{\mathsf{hmv}} \forall\{a\}.\, a \to a$	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 variables
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall a.\, a \to \mathit{Int}$	Can instantiate only a *tail* of the specified variables
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall\{a,b\}.\, a \to b$	Variables can be regeneralized
$\forall a, b.\, a \to b \leq_{\mathsf{hmv}} \forall\{b\}.\, \mathit{Int} \to b$	Because of the right-to-left nature of HMV_INSTS, must regeneralize
$\forall a, b.\, a \to b \not\leq_{\mathsf{hmv}} \forall b.\, \mathit{Int} \to b$	Known variables 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

Figure 5. Examples of HMV subsumption relation

(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 $(e : \upsilon)$ allows us to put a type annotation on an expression. These type annotations are specified polytypes υ and must not contain any free type variables. (We lift this restriction in Section 6.1.) We do not allow annotation by type schemes σ, with quantified generalized variables: if the user writes the type, all quantified variables are considered specified.

4.2.2 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 Figure 3. 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.

Type annotations, typed with HMV_ANNOT, allow expressions to be assigned a specified polytype $\upsilon = \forall\overline{a}.\,\tau$. The rule checks to make sure υ is closed and then types the expression e at type τ. Of course, in the $\Gamma \vdash_{\mathsf{hmv}} e : \tau$ premise, the variables $\overline{a}$ still (perhaps) appear in τ, but they are no longer quantified. 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 $\overline{a}$ and that we can assign e the type $\forall\overline{a}.\,\tau$. This is, in effect, *specified* generalization.

The relation $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$ (Figure 3) 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 the specified variables.

However, it is safe to allow *some* instantiation of specified variables as part of subsumption, as in ruleHMV_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 Figure 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 $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$.*

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 a context generalization lemma, stated and proven in Appendix B, which states that we can generalize types in the context without affecting typability. Note that $\forall\overline{a}.\,\tau \leq_{\mathsf{hmv}} \forall\{\overline{a}\}.\,\tau$.

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

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

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 a system with visible types retains the principal types property, defined with respect to the enhanced subsumption relation $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$.

Theorem 3 (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.

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

$$\frac{x{:}\forall\{\overline{a}\}.\,\tau \in \Gamma}{\Gamma \vdash_{\mathsf{c}} x : \tau[\overline{\tau}/\overline{a}]}\quad \text{C\_VAR}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{c}} e : \tau_2}{\Gamma \vdash_{\mathsf{c}} \lambda x.\, e : \tau_1 \rightarrow \tau_2}\quad \text{C\_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_{\mathsf{c}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{c}} e_1\ e_2 : \tau_2}\quad \text{C\_APP}$$

$$\frac{}{\Gamma \vdash_{\mathsf{c}} n : \mathit{Int}}\quad \text{C\_INT}$$

$$\frac{\Gamma \vdash^{gen}_{\mathsf{c}} e : \sigma \qquad \Gamma, x{:}\sigma \vdash_{\mathsf{c}} e_2 : \tau_2}{\Gamma \vdash_{\mathsf{c}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2}\quad \text{C\_LET}$$

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

$$\frac{\overline{a} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Gamma) \qquad \Gamma \vdash_{\mathsf{c}} e : \tau}{\Gamma \vdash^{gen}_{\mathsf{c}} e : \forall\{\overline{a}\}.\,\tau}\quad \text{C\_GEN}$$

We use $\mathit{ftv}(\sigma)$ to mean the free type variables of a type scheme σ. We lift this to work on contexts: $\mathit{ftv}(\overline{x{:}\sigma}) = \bigcup \mathit{ftv}(\sigma_i)$.

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

In particular, the rules HM_GEN and HM_SUB can appear at any point in a typing derivation.

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 Figure 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 non-deterministic 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 $\mathcal{W}$ guesses 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 (Figure 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 expression 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.

$\boxed{\Gamma \vdash_{\mathsf{v}} e : \tau}$

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{v}} e : \tau_2}{\Gamma \vdash_{\mathsf{v}} \lambda x.\, e : \tau_1 \rightarrow \tau_2}\quad \text{V\_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{v}} e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_{\mathsf{v}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{v}} e_1\ e_2 : \tau_2}\quad \text{V\_APP}$$

$$\frac{}{\Gamma \vdash_{\mathsf{v}} n : \mathit{Int}}\quad \text{V\_INT}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{*}_{\mathsf{v}} e : \forall \overline{a}.\,\tau \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{v}} e : \tau[\overline{\tau}/\overline{a}]}\quad \text{V\_INSTS}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon}$

$$\frac{x{:}\forall\{\overline{a}\}.\,\upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{v}} x : \upsilon[\overline{\tau}/\overline{a}]}\quad \text{V\_VAR}$$

$$\frac{\Gamma \vdash^{gen}_{\mathsf{v}} e_1 : \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{v}} e_2 : \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{v}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \upsilon_2}\quad \text{V\_LET}$$

$$\frac{\tau \text{ closed} \qquad \Gamma \vdash^{*}_{\mathsf{v}} e : \forall a.\,\upsilon}{\Gamma \vdash^{*}_{\mathsf{v}} e\ @\tau : \upsilon[\tau/a]}\quad \text{V\_TAPP}$$

$$\frac{\upsilon \text{ closed} \qquad \upsilon = \forall \overline{a}.\,\tau \qquad \Gamma \vdash_{\mathsf{v}} e : \tau}{\Gamma \vdash^{*}_{\mathsf{v}} (e : \upsilon) : \upsilon}\quad \text{V\_ANNOT}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{v}} e : \tau \\ \text{no other rule matches}\end{array}}{\Gamma \vdash^{*}_{\mathsf{v}} e : \tau}\quad \text{V\_MONO}$$

$\boxed{\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma}$

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

Figure 7. Typing rules for System V

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 quantification in a specified polytype.

The first set of rules in Figure 7, as before, infers a monotype for the expression. The premises of the rule V_ABS uses this 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) allow e to be assigned a specified polytype. Note that the premise of rule V_TAPP uses this judgment.

System V's V_VAR rule is like System C's C_VAR rule: 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 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:

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

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 expression. We prove this correspondence using the following two theorems.

Theorem 4 (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 5 (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 Appendix C.

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 Figure 7 to an algorithm. In Appendix D, 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 3). Furthermore, Algorithm $\mathcal{V}$ is guaranteed to terminate, yielding this theorem

Theorem 6. *Type-checking System V is decidable.*

6. Higher-rank type systems

We now extend the design of System HMV to include two extensions of the Hindley-Milner type system: *higher-rank polymor-*

The grammar for B extends that for System HMV (Figure 4):

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

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

$$\frac{\mathit{ftv}(\upsilon) \subseteq \Gamma}{\Gamma \vdash \upsilon}\ \text{TY\_SCOPED}$$

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

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \sigma}\ \text{B\_VAR}$$

$$\frac{\Gamma, x{:}\tau \vdash_{\mathsf{b}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Rightarrow \tau \to \upsilon}\ \text{B\_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash_{\mathsf{b}} e_1\, e_2 \Rightarrow \upsilon_2}\ \text{B\_APP}$$

$$\frac{}{\Gamma \vdash_{\mathsf{b}} n \Rightarrow \mathit{Int}}\ \text{B\_INT}$$

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

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma \qquad a \notin \mathit{vars}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall \{a\}.\, \sigma}\ \text{B\_GEN}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{b}} \sigma_2}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_2}\ \text{B\_SUB}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash_{\mathsf{b}} e\, @\tau \Rightarrow \upsilon[\tau/a]}\ \text{B\_TAPP}$$

$$\frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}.\, \phi \qquad \Gamma, \overline{a} \vdash_{\mathsf{b}} e \Leftarrow \phi}{\Gamma \vdash_{\mathsf{b}} (e : \upsilon) \Rightarrow \upsilon}\ \text{B\_ANNOT}$$

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

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2}\ \text{B\_DABS}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \Leftarrow \upsilon}\ \text{B\_DLET}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon \qquad a \notin \mathit{vars}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \forall a.\, \upsilon}\ \text{B\_SKOL}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{dsk}} \upsilon_2}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}\ \text{B\_INFER}$$

Figure 8. Grammar and typing rules for System B

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

$$\frac{}{\tau \leq_{\mathsf{b}} \tau} \quad \text{B\_REFL}$$

$$\frac{\upsilon_3 \leq_{\mathsf{b}} \upsilon_1 \qquad \upsilon_2 \leq_{\mathsf{b}} \upsilon_4}{\upsilon_1 \to \upsilon_2 \leq_{\mathsf{b}} \upsilon_3 \to \upsilon_4} \quad \text{B\_FUN}$$

$$\frac{\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \phi_2}{\forall \overline{a}, \overline{b}.\, \phi_1 \leq_{\mathsf{b}} \forall \overline{a}.\, \phi_2} \quad \text{B\_INSTS}$$

$$\frac{\upsilon_1[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon_2 \qquad \overline{b} \notin ftv(\forall \{\overline{a}\}.\, \upsilon_1)}{\forall \{\overline{a}\}.\, \upsilon_1 \leq_{\mathsf{b}} \forall \{\overline{b}\}.\, \upsilon_2} \quad \text{B\_INSTG}$$

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

$$\frac{}{\tau \leq^*_{\mathsf{dsk}} \tau} \quad \text{DSK\_REFL}$$

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

$\boxed{\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2}$ Subsumption, with deep skolemization

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

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

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

Figure 9. Subsumption relations for System B

phism and *scoped type variables*. The former allows function parameters to be used at multiple types, whereas the latter brings type variables into scope to be used in type annotations. Incorporating these extensions shows the generality of our work. Although these extensions come with their own complexities, there are no unpleasant interactions in the introduction of visible type application.

In fact, there is synergy between higher-rank polymorphism and visible type application. GHC supports higher-rank polymorphism [16, 23] through the use of programmer annotations. As in visible type application, this type system feature is enabled only for polytypes that have been specified through some type annotation.

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

```
let foo = λf → (f 3, f True) in foo id
```

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

```
let foo :: (∀ a. a → a) → (Int, Bool)
    foo = λf → (f 3, f True)
in foo id
```

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

```
let foo :: (∀ a. a → a) → (Int → Int, Bool)
    foo = λf → (f @Int, f True)
in foo id
```

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

```
pair :: ∀ a. a → ∀ b. b → (a, b)
pair = λx y → (x, y)
```

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

```
bar = pair 2 @Bool
  -- bar inferred to have type Bool → (Int, Bool)
```

In the rest of this section, we provide the technical details of these language features and discuss their interactions. As above, we start with a declarative specification of the type system, and then discuss its implementation through an equivalent syntax-directed variant. The syntax-directed system studied here is the basis of our implementation in GHC.

6.1 System B: Declarative specification

Figures 8 and 9 show the syntax and typing rules of System B, a declarative bidirectional type system, supporting predicative higher-rank polymorphism, visible type application, and scoped type variables. This declarative type system itself is a novel contribution of this work. Although it is based on the type system studied by Peyton Jones et al. [23], that work uses only a syntax-directed system to describe bidirectional propagation of higher-rank types.

System B is defined by two mutually recursive judgments, $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$ and $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$, specifying when types are synthesized and checked, respectively. In the first judgment, the type σ is an output of the system, whereas in the second, the type υ must be provided along with Γ and e. This system propagates specified type information through the abstract syntax tree via its $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$ judgment.

Although this system is *bidirectional*, we also claim that it is *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.

Basic features of System B System B shares the same expression language of Systems HMV and V, retaining visible type application and type annotations. However, types in System B 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 only over *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. Typing contexts Γ may now contain type variables; this change is used to implement scoped type variables. The function $vars(\Gamma)$ calculates all type variables that occur in Γ, including both the declared scoped type variables and the free type variables in typing assumptions.

Type synthesis The synthesis judgment is identical to the typing judgment for HMV if we ignore direction arrows. This is unsurpris-

$\forall a.\, a \to \forall b.\, b \to b \ \leq_{\mathsf{b}}\ \mathit{Int} \to \mathit{Bool} \to \mathit{Bool}$	Can instantiate non-top-level variables
$\forall a.\, a \to \forall b.\, b \to b \ \leq_{\mathsf{b}}\ \mathit{Int} \to \forall b.\, b \to b$	The levels are independent; not all variables must be instantiated
$\forall a.\, a \to \forall b.\, b \to b \ \leq_{\mathsf{b}}\ \forall a.\, a \to \mathit{Bool} \to \mathit{Bool}$	The levels are independent; we can skip a top-level quantifier
$(\mathit{Int} \to \mathit{Int}) \to \mathit{Bool} \ \leq_{\mathsf{b}}\ (\forall a.\, a \to a) \to \mathit{Bool}$	$\leq_{\mathsf{b}}$ supports contravariant instantiation through higher-rank types
$\mathit{Int} \to \forall a, b.\, a \to b \ \not\leq_{\mathsf{b}}\ \mathit{Int} \to \forall b.\, \mathit{Bool} \to b$	Specified variables are instantiated from the right
$\mathit{Int} \to \forall a.\, a \to a \ \not\leq_{\mathsf{b}}\ \forall a.\, \mathit{Int} \to a \to a$	Cannot move quantifiers for specified variables
$\mathit{Int} \to \forall a, b.\, a \to b \leq_{\mathsf{dsk}} \mathit{Int} \to \forall b.\, \mathit{Bool} \to b$	$\leq_{\mathsf{dsk}}$ can instantiate specified variables in any order
$\mathit{Int} \to \forall a.\, a \to a \leq_{\mathsf{dsk}} \forall a.\, \mathit{Int} \to a \to a$	Specified quantification can move with $\leq_{\mathsf{dsk}}$
$(\mathit{Int} \to \forall b.\, b \to b) \to \mathit{Int} \leq_{\mathsf{dsk}} (\forall a, b.\, a \to b \to b) \to \mathit{Int}$	Out-of-order instantiation works contravariantly through arrows
$\forall \{a\}.\, a \to a \leq_{\mathsf{dsk}} \forall a.\, a \to a$	$\leq_{\mathsf{dsk}}$ ignores the distinction between specified and generalized variables

Figure 10. Examples of B subsumption, with both relations

ing, 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 type information toward the leaves of an expression.

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 Figure 9 is one of two subsumption relations that appear in the type system. 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 for specified type variables. We sometimes say that this judgment enables *inner instantiation* because instantiations are not restricted to top level. Figure 10 provides examples of this relation.

Type checking The checking rules allow the higher-rank type system to take advantage of specified polytypes. This happens in two different ways.

Rule B_DABS is the key rule of the checking judgment. When we have propagated a type $\upsilon_1 \to \upsilon_2$ for an expression $\lambda x.\, e$, B_DABS uses the type υ_1 as x's type when checking e. This is the only place where we can type a function with a higher-rank type. Note that the synthesis rule B_ABS uses a monotype for the type of x.[9]

Rule B_INFER uses the stronger of the two subsumption relations $\leq_{\mathsf{dsk}}$, shown at the bottom of Figure 9 and with examples in Figure 10. This rule appears at precisely the spot in the derivation where a specified type from synthesis mode meets the specified type from checking mode. This relation, called *deep skolemization*, we take directly from prior work [23]. It subsumes $\leq_{\mathsf{b}}$ (that is, $\sigma_1 \leq_{\mathsf{b}} \upsilon_2$ implies $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$) and provides a full subsumption relationship for higher-rank types.

For brevity, we don't explain the details of this relation here, instead referring readers to Peyton Jones et al. (Section 4.6) [23] 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 [20] instead. We present the system with deep skolemization for backwards compatibility with GHC. See Appendix G for a discussion of this alternative.

Rule B_SKOL skolemizes one variable – that is, if we are propagating a type quantified over a (specified) variable a, we continue propagating without the quantification. Like other steps that skolemize, this turns the quantified variable effectively into a type constant. This rule is necessary in order to remove any outer quantification before rule B_DABS can apply. Note that when we talk about skolemization, we are considering reading the rule "bottom-to-top" – the order of the checking propagation. If we read the rule "top-to-bottom", then this is a simple $\forall$-introduction, or generalization, rule.

The remaining checking rule, B_DLET, simply propagates type information into the body of a **let**-expression.

Scoped type variables Scoped type variables increase the expressiveness of our system; they were necessary for both examples in Section 2. Therefore, System B relaxes the HMV restriction that type annotations and instantiations must be closed. Instead, type annotations are allowed to introduce type variables, which are then in scope inside the annotation.

This behavior matches that of GHC [21]. For example, the type annotation on *const* below introduces the type variables *a* and *b* that can then be used in the definition of *const*.

$$\mathit{const} = (\lambda x\ y \to (x :: a) :: \forall\ a\ b.\ a \to b \to a)$$

System B only introduces type variables in B_ANNOT, and there only variables quantified at the top level.

An alternative design would be to introduce scoped type variables in B_SKOL instead of B_ANNOT. (Note that the skolemized variables are not brought into scope in that rule). However, we deliberately avoided this design because it leads to strange scoping behavior, as we describe in Appendix G.

6.2 System SB: Syntax-directed Bidirectional type checking

System B is a declarative type system and not an algorithm. Figure 11 shows the higher rank analogue of System V, called System SB. As with other syntax-directed systems, the form of the rule conclusions resolve the order in which the rules must be applied. System C shows how to restrict generalization to happen only in **let**-bindings; this treatment is retained through all of the systems in this paper. System V shows how to restrict instantiation: instantiate generalized variables eagerly at variable usage sites, and instantiate specified variables lazily, on demand. System SB must now fix the new declarative rule that can apply anywhere: skolemization. System SB requires all skolemization to occur first when checking: SB_DEEPSKOL is the *only* rule in the $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$ judgment, and is also the entry point for the non-skolemizing $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ judgment.[10]

The interaction between rule SB_DEEPSKOL and SB_INFER is subtle, due to the use of deep skolemization as our higher-

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

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

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi}$ Synthesis for ϕ-types

$$\frac{\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \to \upsilon}\ \text{SB\_ABS}$$

$$\frac{}{\Gamma \vdash_{\mathsf{sb}} n \Rightarrow \mathit{Int}}\ \text{SB\_INT}$$

$$\frac{\begin{array}{l}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall \overline{a}.\, \phi \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi[\overline{\tau}/\overline{a}]}\ \text{SB\_INSTS}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}$ Synthesis for υ-types

$$\frac{x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \upsilon[\overline{\tau}/\overline{a}]}\ \text{SB\_VAR}$$

$$\frac{\Gamma \vdash_{\mathsf{sb}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash^{*}_{\mathsf{sb}} e_1\ e_2 \Rightarrow \upsilon_2}\ \text{SB\_APP}$$

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{sb}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \upsilon_2}\ \text{SB\_LET}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} e\ @\tau \Rightarrow \upsilon[\tau/a]}\ \text{SB\_TAPP}$$

$$\frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}.\, \phi \qquad \Gamma, \overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi}{\Gamma \vdash^{*}_{\mathsf{sb}} (e : \upsilon) \Rightarrow \upsilon}\ \text{SB\_ANNOT}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi \\ \text{no other rule matches}\end{array}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \phi}\ \text{SB\_PHI}$$

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

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

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho}$ Checking a ρ-type

$$\frac{\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \rho_2}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \to \rho_2}\ \text{SB\_DABS}$$

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

$$\frac{\begin{array}{l}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{dsk}} \rho_2 \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho_2}\ \text{SB\_INFER}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}$ Checking a υ-type

$$\frac{\begin{array}{l}\mathit{prenex}(\upsilon) = \forall \overline{a}.\, \rho \\ \overline{a} \notin \mathit{vars}(\Gamma) \qquad \Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho\end{array}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}\ \text{SB\_DEEPSKOL}$$

Figure 11. Syntax-directed bidirectional type system

rank subsumption relation. Note that SB_DEEPSKOL does not directly match up with B_SKOL. It doesn't just skolemize the top-level quantified variables; it skolemizes *all* positively quantified variables. For example, if the algorithm is checking against type $\forall a.\, a \to \forall b.\, b \to a$, it will skolemize both a and b, pushing in the type $a \to b \to a$. In fact, the post-skolemization checking judgment $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$, requires that the provided type be a ρ-type – one with no quantifiers to the right of arrows.

Deep skolemization is necessary in SB_DEEPSKOL because SB_INFER uses the $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon$ synthesis judgment in its premise, instead of the $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma$ judgment. This decision to avoid generalization was forced by GHC, where generalization is intricately tied into its treatment of **let**-bindings and not supported for arbitrary expressions. Compare SB_INFER with B_INFER, whose premise synthesizes a σ-type. This difference means that, in the syntax-directed system, we require more instantiations in the typing derivation above the SB_INFER rule. If the checked type were not deeply skolemized, certain inner-quantified variables would be unavailable for instantiation. For an illuminating example, see Figure 12.

Properties of System B and SB System SB faithfully implements System B.

Lemma 7 (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 8 (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* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$.

Furthermore, we also show that System B is flexible with respect to the instantiation relation $\leq_{\mathsf{b}}$. As in System HMV, this result implies that making generalized variables into specified variables does not disrupt types.

Lemma 9 (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 Appendix F.

7. 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. Below, we describe describe interactions between visible type application and other features of GHC.

7.1 Case expressions

Typing rules for case analysis and **if**-expressions require that all branches have the same type. But what sort of type should that be? For example, consider the following expression

if *condition* **then** *id* **else** $(\lambda x \to x)$

Here, *id* has a specified polytype of $\forall\ a.\ a \to a$, but the expression $\lambda x \to x$ does not. To make this code type check, GHC must find a common type for both branches.

One option would be to generalize the type of $\lambda x \to x$ and then choose $\forall\ a.\ a \to a$ as the common supertype of itself and $\forall\{a\}.\, a \to a$. However, that may not be possible in general, as there may not always be a common instance of both types.

Assume $\Gamma = x{:}\forall\{a\}.\, \mathit{Int} \to a \to a$. We wish to type-check the expression $(x{:}\mathit{Int} \to \forall a.\, a \to a)$. Here is a valid derivation in System B:

$$\dfrac{\dfrac{x{:}\forall\{a\}.\, \mathit{Int} \to a \to a \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \forall\{a\}.\, \mathit{Int} \to a \to a}\ \text{B\_VAR} \quad \dfrac{\dots}{\forall\{a\}.\, \mathit{Int} \to a \to a \leq_{\mathsf{dsk}} \mathit{Int} \to \forall a.\, a \to a}\ \text{B\_INSTG}}{\dfrac{\Gamma \vdash_{\mathsf{b}} x \Leftarrow \mathit{Int} \to \forall a.\, a \to a}{\Gamma \vdash_{\mathsf{b}} (x : \mathit{Int} \to \forall a.\, a \to a) \Rightarrow \mathit{Int} \to \forall a.\, a \to a}\ \text{B\_ANNOT}}\ \text{B\_INFER}$$

Here is a valid derivation in System SB:

$$\dfrac{\dfrac{\dfrac{\dfrac{x{:}\forall\{a\}.\, \mathit{Int} \to a \to a \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \mathit{Int} \to a \to a}\ \text{SB\_VAR} \quad \dfrac{\dfrac{}{\mathit{Int} \to a \to a \leq^{*}_{\mathsf{dsk}} \mathit{Int} \to a \to a}\ \text{DSK\_REFL}}{\mathit{Int} \to a \to a \leq_{\mathsf{dsk}} \mathit{Int} \to a \to a}\ \text{DSK\_INST}}{\Gamma \vdash_{\mathsf{sb}} x \Leftarrow \mathit{Int} \to a \to a}\ \text{SB\_INFER}}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Leftarrow \mathit{Int} \to \forall a.\, a \to a}\ \text{SB\_DEEPSKOL}}{\dfrac{\Gamma \vdash^{*}_{\mathsf{sb}} (x : \mathit{Int} \to \forall a.\, a \to a) \Rightarrow \mathit{Int} \to \forall a.\, a \to a}{\Gamma \vdash^{gen}_{\mathsf{sb}} (x : \mathit{Int} \to \forall a.\, a \to a) \Rightarrow \mathit{Int} \to \forall a.\, a \to a}\ \text{SB\_GEN}}\ \text{SB\_ANNOT}$$

Note the deep skolemization in this derivation. If we did only a shallow skolemization at the point we use SB_DEEPSKOL, then a would not be skolemized. Accordingly, it would be impossible to instantiate the type of x with a in the use of the SB_VAR rule.

Figure 12. An example of why deep skolemization in SB_DEEPSKOL is necessary

Instead, following prior work [23], we require that **if** and **case** expressions synthesize monotypes. Accordingly, the type checker instantiates the type *id* above before unification.

Note that specified polytypes are still available for type *checking* because we know the type that each branch should have. For example, the following declaration is accepted

$\mathit{checkIf} :: \mathit{Bool} \to (\forall\ a.\ a \to a) \to (\mathit{Bool}, \mathit{Int})$
$\mathit{checkIf}\ b = \textbf{if}\ \mathit{True}$
  $\textbf{then}\ \lambda f \to (f\ \mathit{True}, f\ 5)$
  $\textbf{else}\ \lambda f \to (f\ \mathit{False}, f\ 3)$

7.2 Imported functions

The key requirement of specified polytypes is that type variables are fixed and known to programmers. But, when is this the case?

In the design of our implementation, we considered the possibility that *all* imported functions could meet this requirement. This would allow visible type application for any imported function, whether or not it was originally supplied with a type annotation. This decision is justified: programmers can use tools (such as `ghci`'s `:browse` command) to discover the types. This decision also places the least burden on programmers, as library authors need not think about visible type application when deciding whether to specify the types of their functions.

However, this design is also fragile. By allowing all imported functions to be visibly instantiated, the ordering of type variable quantification is now part of the specification of the function. Perhaps worse, there is no guarantee that this ordering will stay the same from one version of the compiler to the next.

Furthermore, it is also unnecessary. Haskell library authors already do put type signatures on many of their exported functions. For functions exported without a type signature, clients may easily add their own type specifications by rebinding imported functions.

We thus chose the conservative design. Only imported functions with type signatures are considered to have specified types.[11]

7.3 Partial type signatures

Partial type signatures [29] are a recent addition to GHC, allowing users to leave *wildcards* in types, allowing GHC to infer those parts of a type. Wildcards can appear in visible type arguments, allowing users to skip types GHC can infer. For example, if f has type $\forall\ a\ b.\ a \to b \to (a, b)$, then we can write $f\ @\_\ @[\mathit{Int}]\ \mathit{True}\ [\,]$ to let GHC infer that a should be *Bool* but to visibly instantiate b to be $[\mathit{Int}]$. The existing partial type signatures machinery simply fills in the wildcard by unification, as it does when wildcards appear in type signatures.

7.4 Futher extensions to visible type application

Our implementation also gives us the chance to explore two related extensions in future work.

Visible type binding in patterns Consider the following GADT

$\textbf{data}\ G\ a\ \textbf{where}$
  $\mathit{MkG} :: \forall\ b.\ G\ (\mathit{Maybe}\ b)$

When pattern-matching on a value of type $G\ a$ to get the constructor *MkG*, we would want a mechanism to bind a type variable to b, the argument to *Maybe*. A visible type pattern makes this easy

$\textbf{case}\ g\ \textbf{of}$
  $\mathit{MkG}\ @b \to \ldots$

The type variable b may now be used as a scoped type variable in the body of the match.

Visible kind application The following function is kind-polymorphic [30]

$\mathit{pr} :: \forall\ (a :: k_1 \to k_2)\ (b :: k_1).\ \mathit{Proxy}\ (a\ b) \to \mathit{Proxy}\ a$
$\mathit{pr}\ \_ = \mathit{Proxy}$

Yet, even with our extension, we cannot instantiate the kind parameters k_1 and k_2 visibly; all kind variables are treated as generalized variables. We expect to address this deficiency in future work.

8. Related work and Conclusions

Implicit arguments in dependently-typed languages Languages such as Coq [6], Agda [19], 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

[11] If the type signature does not include an explicit $\forall$ listing the type variables, we use the order as they appear in the user-supplied type signature.

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.

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 proposed in this paper. Instead, the user must rely on type annotations to control instantiations.

Predicative, higher-rank type systems As we have already indicated, System B is directly inspired by GHC's design for higher-rank types [23]. However, in this work we have pushed the design further, clarifying the treatment of scoped type variables and providing a declarative specification for the type system.

Our work is also related to recent work on using a bidirectional type system for higher-rank polymorphism by Dunfield and Krishnaswami [9], called DK below. There are a few differences between the DK system and System B. The most significant difference is that the DK system never generalizes. All polymorphic types in their system would be considered specified. As a result, their system cannot infer prenex polymorphism: a function must have a type annotation to be polymorphic. Furthermore, System B includes two forms of subsumption. The more flexible relation $\leq_{\mathsf{dsk}}$ requires two specified polytypes so is only available at mode switches. DK also includes this relation, though a weaker version. However, System B also includes implicit subsumption $\leq_{\mathsf{b}}$, which does not have an analogue in the DK system. Instead, the DK system requires an "application judgment form" for instantiation. Finally, DK uses a different algorithm for type checking than the one proposed in this work; though like this paper, it defers instantiations of specified polymorphism. An extended comparison with the DK system appears in Appendix G.

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. In particular, 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 and Conor McBride for helpful discussion.

References

[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] 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.

[10] 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.

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

[12] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Workshop on Types in Languages Design and Implementation*. ACM, 2003.

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Draft, 2004. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/`.

[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] Dimitrios Vytiniotis, Stephanie C. Weirich, and Simon Peyon Jones. Practical type inference for arbitrary-rank types: Technical appendix.

Technical Report MS-CIS-05-14, University of Pennsylvania, 2005. URL http://repository.upenn.edu/cis_reports/58/.

[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.

[30] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, TLDI '12. ACM, 2012.

A. Extended examples using visible types

In this section we present two longer examples that benefit from the addition of visible type application. The second expands and explains the code presented in Section 2.

A.1 Deferring constraints to runtime

Recent work [2] uses the following definition to enable mixing static and dynamic typing in order to implement information-flow control in Haskell:[12]

```
class Deferrable (c :: Constraint) where
  assume :: ∀ a. Proxy c → (c ⇒ a) → a
```

The parameter to the class *Deferrable* is a constraint kind – that is, the kind classifying constraints that appear to the left of ⇒. For example, *Show a* is a *Constraint*. The idea behind *Deferrable* is that, if a constraint is deferred, the program calculates *at runtime* whether or not the constraint holds.

Let's consider deferring an equality constraint, written $\tau_1 \sim \tau_2$ in Haskell. Equality constraints are ordinary constraints; in particular, they have kind *Constraint* and can thus be deferred. However, if we have some type variable *a* and wish to check if *a* is, say, *Bool* at runtime, we need runtime type information. Haskell's *Typeable* feature [12] implements runtime type information. If we have a function

```
woozle :: Typeable a ⇒ a → a
```

then runtime information identifying the type *a* is available at runtime, in the body of *woozle*.

Putting this all together, it seems reasonable to defer an equality constraint between two types if we have runtime type information for both of them:

```
instance (Typeable a, Typeable b)
    ⇒ Deferrable (a ∼ b) where ...
```

However, to implement *assume*, we need one more definition.

A.1.1 Propositional equality: :∼:

Recent standard libraries shipped with GHC contain the following datatype:

```
data a :∼: b where
  Refl :: a :∼: a
```

This datatype implements propositional equality. If you have a value *pf* :: τ_1 :∼: τ_2, that is a proof that types τ_1 and τ_2 are equal. Pattern matching on *pf* reveals this equality to GHC's type-checker, which can then use it in a pattern match:

```
boolCast :: (a :∼: Bool) → a → Bool
boolCast pf b = case pf of Refl → b
```

A.1.2 Runtime cast

The *Typeable* feature uses :∼: in an important function:

```
eqT :: (Typeable a, Typeable b) ⇒ Maybe (a :∼: b)
```

Given runtime type information for *a* and *b*, this function conditionally provides a proof that *a* and *b* are equal. The *eqT* function, in turn, can be used to implement a runtime cast.

We are now ready to assemble the pieces:

```
instance (Typeable a, Typeable b)
    ⇒ Deferrable (a ∼ b) where
```

[12] Much of this example – including its use of deferring equality constraints – appears in Buiras et al. [2]. However, our use of visible type application in this example is our own contribution, novel in this paper.

```
  assume _ x = case eqT :: Maybe (a :~: b) of
    Just Refl → x
    Nothing  → error "type error!"
```

A.1.3 Making assumptions

Suppose we are working a list type that tracks whether it has surely one element, or whether there is an unknown length.[13] Here are the relevant definitions:

```
data Flag = Branched    -- 0 or more elements
          | Unbranched  -- exactly 1 element
data List a (b :: Flag) = ...
the :: List a Unbranched → a
the = ...
```

In some places, it is hard to arrange for the type system to ascertain that a list is *Unbranched*, and calling *the* is impossible. However, with *Deferrable*, we can get around that pesky static type system:

```
unsafeThe :: ∀ a b. Typeable b ⇒ List a b → a
unsafeThe ℓ
  = assume (Proxy :: Proxy (b ∼ Unbranched))
      (the ℓ)
```

The call to *assume* means that *the* ℓ is type-checked in an environment where the constraint $b \sim Unbranched$ is assumed. The call *the* ℓ then type-checks without a problem.

A.1.4 Deferring errors with visible type application

This last snippet of code assumes a constraint, and the only way of specifying the constraint is via a *Proxy*. This is what visible type application can ameliorate. Let's rewrite this example with visible type application.

```
class Deferrable (c :: Constraint) where
  assume :: ∀ a. (c ⇒ a) → a
instance (Typeable a, Typeable b)
  ⇒ Deferrable (a ∼ b) where
  assume x = case eqT @a @b of
    Just Refl → x
    Nothing  → error "type error!"
unsafeThe :: ∀ a b. Typeable b ⇒ List a b → a
unsafeThe ℓ = assume @(b ∼ Unbranched) (the ℓ)
```

We have used visible type applications in two places here. One is to fix the type of the call to *eqT*. Because we immediately pattern-match on this result, GHC has no way of inferring the types at which to use *eqT*. In the previous version of this example, it was necessary to write *eqT* :: *Maybe* (*a* :∼: *b*) here. This annotation is noisy, because we care only about the *a* and the *b* – the *Maybe* and :∼: bits are fixed and add no information. It is easy to imagine more complex cases where the noise far outstrips the signal.

The second use of visible type application is in the definition and call of *assume*, where no *Proxy* argument is now necessary. Once again, this has cleaned up our code and drastically reduced noise.

Dependently-typed programming with Proxy Dependently-typed programming in GHC can require more extensive use of proxies. For example, based on Conor McBride's ICFP 2012 keynote [15], consider a stack-based compiler for a language of boolean expressions. (The entire code for this example is available in the supplementary material.)

```
data Expr :: ⋆where
  Val  :: Bool → Expr
  Cond :: Expr → Expr → Expr → Expr
eval :: Expr → Bool
eval (Val n)          = n
eval (Cond e0 e1 e2) =
  if eval e0 then eval e1 else eval e2
```

Using standard techniques, we can create a *singleton* type for expressions *SExpr* and a type-level function *Eval* that allow the type system to talk about these definitions.

```
eval :: Expr → Bool
eval (Val n) = n
eval (Cond e0 e1 e2) = if eval e0 then eval e1 else eval e2
type family Eval (x :: Expr) :: Bool where
  Eval ('Val n) = n   Eval ('
Cond e0 e1 e2) = If (Eval e0) (Eval e1) (Eval e2)
```

For example, the evaluator for singleton booleans states that it actually calculates the boolean denoted by the expression:

```
sEval :: SExpr e → SBool (Eval e)
sEval (SVal n) = n
sEval (SCond e0 e1 e2) = sIf (sEval e0) (sEval e1) (sEval e2)
```

However, instead of evaluating booleans directly, we would like to compile them to a list of instructions for a stack machine. At the same time, we would like to know that the resulting list of instruction will produce the correct answer when run.

In other words, given a GADT representing instruction lists, that when run will take a stack from its initial configuration to the final configuration:

```
data Inst (initial :: [Bool]) (final :: [Bool]) where
    -- Add a value to the top of the stack
  PUSH :: Sing v → Inst s (v ': s)
    -- Compare the top value on the stack and branch
  IFPOP :: ListInst s st → ListInst s sf
          → Inst (b ': s) (If b st sf)
  -- a list of instructions, also tracking the machine configurations
data ListInst (initial :: [Bool]) (final :: [Bool]) where
  Nil :: ListInst i i
  (:::) :: Inst i j → ListInst j k → ListInst i k
infixr 5 :::
  -- concatenate two lists, composing their stacks
(++) :: ListInst i j → ListInst j k → ListInst i k
Nil ++ ys = ys
(x ::: xs) ++ ys = x ::: (xs ++ ys)
infixr 5 ++
```

We would like to define a compilation function that will create a list of instructions that, when run, will put the evaluation of an expression at the top of the stack.

```
compile :: ∀ (e :: Expr) (s :: [Bool]).
  SExpr e → ListInst s ((Eval e) ': s)
```

The implementation of the compilation function is straightforward in the case of a singleton boolean value. It just pushes that value on the top of an empty stack.

```
compile (SVal y) = PUSH y ::: Nil
```

[13] This example is from real code – just such a list is used within GHC when keeping track of type family axioms from either open [3, 4] or closed [10] type families.

However, the compilation of conditionals runs into difficulties, we would like to use this code, which first compiles the scrutinee, and then appends the branch instruction.

```
compile (SCond se0 se1 se2) =
  compile se0 ++
    IFPOP (compile se1) (compile se2) ::: Nil
```

However, for this code to type check, the compiler needs to know the following conversion fact about if expressions

```
(If (Eval e0) (Eval e1) (Eval e2)) ': vs) :~:
  (If (Eval e0) ((Eval e1) ': vs) ((Eval e2) ': vs))
```

We can "prove" this fact to the compiler, with a helper lemma, called *fact* below. Note, however that in the result of the lemma, the type variables *t* and *f* only appear as arguments to the type-level function *If*. Therefore, unification cannot be used to instantiate these arguments, so the *Proxy* type is necessary.

```
fact :: ∀ t f s b. Sing b → Proxy t → Proxy f → Proxy s
  → ((If b t f) ': s) :~: (If b (t ': s) (f ': s))
fact STrue _ _ _ = Refl
fact SFalse _ _ _ = Refl
```

We can call *fact* in the case for *compile*, by providing the appropriate *Proxy* arguments.

```
compile (SCond se0 (se1 :: Sing e1) (se2 :: Sing e2)) =
  case fact (sEval se0) (Proxy :: Proxy (Eval e1))
    (Proxy :: Proxy (Eval e2)) (Proxy :: Proxy s) of
    Refl → compile se0 ++
      IFPOP (compile se1) (compile se2) ::: Nil
```

Note, that in our definition of *fact* above, we have made the argument *s* be specified via *Proxy*, even though it doesn't technically need to to be because it appears outside of the *If* in the type. GHC will also accept this alternative *fact* ' that does not include a *Proxy s* argument.

```
fact ' :: ∀ t f s b. Sing b → Proxy t → Proxy f
  → ((If b t f) ': s) :~: (If b (t ': s) (f ': s))
fact 'STrue _ _ = Refl
fact 'SFalse _ _ = Refl
```

However, that version of *fact* is even more difficult to use. Because the result of *fact* ' is used as the argument of *GADT* pattern matching, GHC cannot use unification to resolve type variables in this type. Instead, to make this code type check, we require an even more extensive type annotation:

```
compile (SCond (se0 :: Sing e0)
  (se1 :: Sing e1) (se2 :: Sing e2)) =
  case (fact '(sEval se0)
        (Proxy :: Proxy (Eval e1))
        (Proxy :: Proxy (Eval e2)) ::
    ((If (Eval e0) (Eval e1) (Eval e2)) ': s) :~:
      (If (Eval e0) ((Eval e1) ': s) ((Eval e2) ': s))) of
    Refl → compile se0 ++
      IFPOP (compile se1) (compile se2) ::: Nil
```

In the presence of visible type application, we would like to avoid the proxies all together:

```
fact :: ∀ t f s b. Sing b →
    ((If b t f) ': s) :~: (If b (t ': s) (f ': s))
fact STrue = Refl
fact SFalse = Refl
```

and supply the type arguments visibly:

```
compile (SCond se0 (se1 :: Sing e1) (se2 :: Sing e2)) =
  case fact @(Eval e1) @(Eval e2) @s (sEval se0) of
    Refl → compile se0 ++
      IFPOP (compile se1) (compile se2) ::: Nil
```

B. Properties of System HMV

Lemma 10 (Inversion for $\leq_{\mathsf{hmv}}$)**.** *$\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$ if and only if $\sigma_1 = \forall\{\overline{a}_1\}, \overline{b}_2, \overline{b}_1. \tau_1$ and $\sigma_3 = \forall\{\overline{a}_2\}, \overline{b}_2. \tau_2$ where $\tau_1[\overline{\tau}_1/\overline{a}_1][\overline{\tau}'_1/\overline{b}_1] = \tau_2$.*

Proof. By unfolding definitions. □

Lemma 11 (Reflexivity for $\leq_{\mathsf{hmv}}$)**.** *Forall σ, $\sigma \leq_{\mathsf{hmv}} \sigma$*

Proof. By definition. □

Lemma 12 (Transitivity for $\leq_{\mathsf{hmv}}$)**.** *If $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$ and $\sigma_2 \leq_{\mathsf{hmv}} \sigma_3$, then $\sigma_1 \leq_{\mathsf{hmv}} \sigma_3$.*

Proof. Let $\sigma_3 = \forall\{\overline{a}_3\}, \overline{b}_3. \tau_3$. Then, by inversion, we know $\sigma_2 = \forall\{\overline{a}_2\}, \overline{b}_3, \overline{b}_2. \tau_2$ and $\tau_2[\overline{\tau}_2/\overline{a}_2][\overline{\tau}'_2/\overline{b}_2] = \tau_3$. We further know $\sigma_1 = \forall\{\overline{a}_1\}, \overline{b}_3, \overline{b}_2, \overline{b}_1. \tau_1$ and $\tau_1[\overline{\tau}_1/\overline{a}_1][\overline{\tau}'_1/\overline{b}_1] = \tau_2$. Thus, $\tau_1[\overline{\tau}_1/\overline{a}_1][\overline{\tau}'_1/\overline{b}_1][\overline{\tau}_2/\overline{a}_2][\overline{\tau}'_2/\overline{b}_2] = \tau_3$. By the Barendregt convention, we know that $\overline{a}_2$ do not appear in τ_1. Thus we can rewrite as $\tau_1[\overline{\tau}_1[\overline{\tau}_2/\overline{a}]/\overline{a}_1][\overline{\tau}'_1[\overline{\tau}_2/\overline{a}_2]/\overline{b}_1][\overline{\tau}'_2/\overline{b}_2] = \tau_3$. This is enough to finish the derivation via HMV_INSTS and HMV_INSTG. □

Lemma 13 (Substitution in $\leq_{\mathsf{hmv}}$)**.**

1. *If $\upsilon_1 \leq_{\mathsf{hmv}} \upsilon_2$, then $\upsilon_1[\tau/a] \leq_{\mathsf{hmv}} \upsilon_2[\tau/a]$.*
2. *If $\sigma_1 \leq_{\mathsf{hmv}} \sigma_2$, then $\sigma_1[\tau/a] \leq_{\mathsf{hmv}} \sigma_2[\tau/a]$.*

Proof. Immediate. □

Lemma 14 (Context Generalization for HMV)**.** *If $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$ and $\Gamma' \leq_{\mathsf{hmv}} \Gamma$, then $\Gamma' \vdash_{\mathsf{hmv}} e : \sigma$.*

Proof. This is by straightforward induction, with an appeal to HMV_SUB in the variable case (HMV_VAR). □

Proof of Lemma 2 (Extra knowledge). This is a corollary of contexts generalization as $\forall\{\overline{a}\}. \tau \leq_{\mathsf{hmv}} \forall\overline{a}. \tau$. □

C. Proofs about System V

Proof of Soundness of V against HMV (Theorem 4). By induction on the appropriate derivation. Most cases follow directly via induction.

Case V_INSTS This case follows via induction and HMV_SUB using the fact that $\forall\overline{a}. \tau \leq_{\mathsf{hmv}} \tau[\overline{\tau}/\overline{a}]$ by HMV_INSTS.

Case V_VAR This case follows via HMV_VAR and HMV_SUB using the fact that $\forall\{\overline{a}\}. \tau \leq_{\mathsf{hmv}} \tau[\overline{\tau}/\overline{a}]$ by HMV_INSTG.

□

Lemma 15 (Context generalization for V)**.**

1. *If $\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon$ and $\Gamma' \leq_{\mathsf{hmv}} \Gamma$, then there exists υ' such that $\Gamma' \vdash^{*}_{\mathsf{v}} e : \upsilon'$ and $\upsilon' \leq_{\mathsf{hmv}} \upsilon$.*
2. *If $\Gamma \vdash_{\mathsf{v}} e : \tau$ and $\Gamma' \leq_{\mathsf{hmv}} \Gamma$, then $\Gamma' \vdash_{\mathsf{v}} e : \tau$*
3. *If $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma$ and $\Gamma' \leq_{\mathsf{hmv}} \Gamma$, then there exists σ' such that $\Gamma' \vdash^{gen}_{\mathsf{v}} e : \sigma'$ and $\sigma' \leq_{\mathsf{hmv}} \sigma$.*

In all cases, the size of resulting derivation is no larger than the size of the input derivation.

Proof. By induction on derivations. Most cases are straightforward; we present the most illuminating cases below:

Case V_VAR:

$$\frac{x{:}\forall\{\overline{a}\}.\,\upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{v}} x : \upsilon[\overline{\tau}/\overline{a}]}\quad \text{V\_VAR}$$

Given $x{:}\forall\{\overline{a}'\}.\,\upsilon' \in \Gamma'$ where $\forall\{\overline{a}'\}.\,\upsilon' \leq_{\mathsf{hmv}} \forall\{\overline{a}\}.\,\upsilon$, we must choose $\overline{\tau}'$ such that $\upsilon'[\overline{\tau}'/\overline{a}'] \leq_{\mathsf{hmv}} \upsilon[\overline{\tau}/\overline{a}]$. Inverting $\leq_{\mathsf{hmv}}$ gives us that $\upsilon'[\overline{\tau}'/\overline{a}'] \leq_{\mathsf{hmv}} \upsilon$. We are thus done by Lemma 13. Note that the size of both derivations is 1.

Case V_TAPP:

$$\frac{\tau \text{ closed} \qquad \Gamma \vdash^{*}_{\mathsf{v}} e : \forall a.\,\upsilon}{\Gamma \vdash^{*}_{\mathsf{v}} e\ @\tau : \upsilon[\tau/a]}\quad \text{V\_TAPP}$$

The induction hypothesis gives us $\Gamma' \vdash^{*}_{\mathsf{v}} e : \upsilon'$ where $\upsilon' \leq_{\mathsf{hmv}} \forall a.\,\upsilon$. By the definition of $\leq_{\mathsf{hmv}}$, υ' must also be quantified over a. We can thus reduce to $\forall a.\,\upsilon'' \leq_{\mathsf{hmv}} \forall a.\,\upsilon$, which reduces to $\upsilon'' \leq_{\mathsf{hmv}} \upsilon$. We must prove that $\upsilon''[\tau/a] \leq_{\mathsf{hmv}} \upsilon[\tau/a]$, which follows directly from $\upsilon'' \leq_{\mathsf{hmv}} \upsilon$ via Lemma 13, and so we are done.

Case V_GEN:

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

The induction hypothesis gives us $\Gamma' \vdash^{*}_{\mathsf{v}} e : \upsilon'$ where $\upsilon' \leq_{\mathsf{hmv}} \upsilon$. By inversion, we know that $\upsilon' = \forall\overline{a}', \overline{b}'.\,\tau_1$ and $\upsilon = \forall\overline{a}'.\,\tau_2$ where $\tau_1[\overline{\tau}'/\overline{b}'] = \tau_2$.

Let $\overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)$ and $\overline{b} = \mathit{ftv}(\upsilon') \setminus \mathit{ftv}(\Gamma')$. We want to show that $\forall\{\overline{b}\}.\,\upsilon' \leq_{\mathsf{hmv}} \forall\{\overline{a}\}.\,\upsilon$. Expanding out, we want to show that $\forall\{\overline{b}\}, \overline{a}', \overline{b}'.\,\tau_1 \leq_{\mathsf{hmv}} \forall\{\overline{a}\}, \overline{a}'.\,\tau_2$, where $\tau_1[\overline{\tau}/\overline{b}] = \tau_2$. Unfolding $\leq_{\mathsf{hmv}}$ shows that we want $\tau_1[\overline{\tau}/\overline{b}][\overline{\tau}'/\overline{b}'] = \tau_2$. Choose $\overline{\tau} = \overline{b}$ and we are done.

□

Lemma 16 (Substitution for V)**.**

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

Proof. By induction, frequently using the Barendregt convention to rename bound variables to avoid coinciding with free variables.

The interesting case is generalization: The premise of this case is $\Gamma \vdash^{*}_{\mathsf{v}} e : \upsilon$ where $\sigma = \forall\{\overline{b}\}.\,\upsilon$ for $\overline{b} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)$. By the Barendregt convention, note that $\overline{b}$ do not contain a.

The induction hypothesis gives us $\Gamma[\tau/a] \vdash^{*}_{\mathsf{v}} e : \upsilon[\tau/a]$. Let $\overline{c} = \mathit{ftv}(\upsilon[\tau/a]) \setminus \mathit{ftv}(\Gamma[\tau/a])$. To use V_GEN to conclude $\Gamma \vdash^{gen}_{\mathsf{v}} e : (\forall\{\overline{c}\}.\,\upsilon)[\tau/a]$, we must show that $\overline{b} = \overline{c}$ and that $\overline{c}$ are not free in τ.

We now have several cases:

a **is free in both** υ **and in** Γ**:** In this case $\mathit{ftv}(\upsilon[\tau/a])$ includes the free type variables of υ, minus a, plus the free variables of τ. Likewise, $\mathit{ftv}(\Gamma[\tau/a])$ includes $\mathit{ftv}(\Gamma)$, minus a, plus the free variables of τ. In each case, the sets that produce $\overline{c}$ are changed by the same variables. Therefore, $\overline{b}$ and $\overline{c}$ must be equal.

a **is free in** υ **but not free in** Γ**:** To be in this case a must be in $\overline{b}$, which cannot happen.

a **is not free in** υ **but is free in** Γ**:** In this case $\overline{b} = \overline{c}$ and we are easily done.

a **is free in neither** υ **nor** Γ**:** In this case the substitution has no effect and we are done.

□

Proof of Completeness (Theorem 5). We proceed by induction on $\Gamma \vdash_{\mathsf{hmv}} e : \sigma$.

Case HMV_VAR: Straightforward, using the types $\overline{a}$ to instantiate the variables $\overline{a}$ in V_VAR. We know these $\overline{a}$ are not free in Γ by the Barendregt convention. It may be the case that generalization quantifies over more variables, i.e. $\overline{a} \subseteq \overline{a}' = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)$, leading to a more general result type. However, that is permitted by the statement of the theorem.

Case HMV_ABS:

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{hmv}} e : \tau_2}{\Gamma \vdash_{\mathsf{hmv}} \lambda x.\, e : \tau_1 \to \tau_2}\quad \text{HMV\_ABS}$$

The induction hypothesis gives us $\Gamma, x{:}\tau_1 \vdash^{gen}_{\mathsf{v}} e : \forall\{\overline{a}\}, \overline{b}.\,\tau_2'$ where $\tau_2 = \tau_2'[\overline{\tau}'/\overline{b}][\overline{\tau}/\overline{a}]$. Inverting $\vdash^{gen}_{\mathsf{v}}$ gives us $\Gamma, x{:}\tau_1 \vdash^{*}_{\mathsf{v}} e : \forall\overline{b}.\,\tau_2'$. We can then use V_INSTS and V_ABS to get $\Gamma \vdash_{\mathsf{v}} \lambda x.\, e : \tau_1 \to \tau_2'[\overline{\tau}'/\overline{b}]$. Generalizing, we get $\Gamma \vdash^{gen}_{\mathsf{v}} \lambda x.\, e : \forall\{\overline{a}, \overline{a}'\}.\,\tau_1 \to \tau_2'[\overline{\tau}'/\overline{b}]$ where the new variables $\overline{a}'$ come from generalizing τ_1 and the $\overline{\tau}'$. We can see that $(\tau_1 \to \tau_2'[\overline{\tau}'/\overline{b}])[\overline{\tau}/\overline{a}] = \tau_1 \to \tau_2$ and so $\forall\{\overline{a}, \overline{a}'\}.\,\tau_1 \to \tau_2'[\overline{\tau}'/\overline{b}] \leq_{\mathsf{hmv}} \tau_1 \to \tau_2$ and we are done.

Case HMV_APP:

$$\frac{\Gamma \vdash_{\mathsf{hmv}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{hmv}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{hmv}} e_1\ e_2 : \tau_2}\quad \text{HMV\_APP}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{v}} e_1 : \forall\{\overline{a}_1\}, \overline{b}_1.\,\tau_{11} \to \tau_{12}$ with $\tau_1 = \tau_{11}[\overline{\tau}_1/\overline{a}_1][\overline{\tau}_1'/\overline{b}_1]$ and $\tau_2 = \tau_{12}[\overline{\tau}_1/\overline{a}_1][\overline{\tau}_1'/\overline{b}_1]$, along with $\Gamma \vdash^{gen}_{\mathsf{v}} e_2 : \forall\{\overline{a}_2\}, \overline{b}_2.\,\tau_{21}$ with $\tau_1 = \tau_{21}[\overline{\tau}_2/\overline{a}_2][\overline{\tau}_2'/\overline{b}_2]$. Inverting $\vdash^{gen}_{\mathsf{v}}$ gives us $\Gamma \vdash^{*}_{\mathsf{v}} e_1 : \forall\overline{b}_1.\,\tau_{11} \to \tau_{12}$ and $\Gamma \vdash^{*}_{\mathsf{v}} e_2 : \forall\overline{b}_2.\,\tau_{21}$.

We now use the Substitution Lemma (Lemma 16) with the substitution $[\overline{\tau}_1/\overline{a}_1]$ on the first of these to yield $\Gamma \vdash^{*}_{\mathsf{v}} e_1 : \forall\overline{b}_1.\,\tau_{11}[\overline{\tau}_1/\overline{a}_1] \to \tau_{12}[\overline{\tau}_1/\overline{a}_1]$. Note that the $\overline{a}_1$ must not be free in Γ, by inversion of $\vdash^{gen}_{\mathsf{v}}$. Similarly, Lemma 16 gives us $\Gamma \vdash^{*}_{\mathsf{v}} e_2 : \forall\overline{b}_2.\,\tau_{21}[\overline{\tau}_2/\overline{a}_2]$.

We can then use V_INSTS on both of these judgments, to show $\Gamma \vdash_{\mathsf{v}} e_1 : \tau_{11}[\overline{\tau}_1/\overline{a}_1][\overline{\tau}_1'/\overline{b}_1] \to \tau_{12}[\overline{\tau}_1/\overline{a}_1][\overline{\tau}_1'/\overline{b}_1]$ and $\Gamma \vdash_{\mathsf{v}} e_2 : \tau_{21}[\overline{\tau}_2/\overline{a}_2][\overline{\tau}_2'/\overline{b}_2]$.

We can now now use V_APP, as the argument type is equal to τ_1, established earlier. Rule V_APP then gives us $\Gamma \vdash_{\mathsf{v}} e_1\ e_2 : \tau_{12}[\overline{\tau}_1/\overline{a}_1][\overline{\tau}_1'/\overline{b}_1]$. This type, as noted earlier, equals τ_2, and so we are done.

Case HMV_INT

$$\frac{}{\Gamma \vdash_{\mathsf{hmv}} n : \mathit{Int}}\quad \text{HMV\_INT}$$

Trivial.

Case HMV_TAPP:

$$\frac{\tau \text{ closed} \qquad \Gamma \vdash_{\mathsf{hmv}} e : \forall a.\,\upsilon}{\Gamma \vdash_{\mathsf{hmv}} e\ @\tau : \upsilon[\tau/a]}\quad \text{HMV\_TAPP}$$

The induction hypothesis (after inverting $\vdash^{gen}_{\mathsf{v}}$) gives us $\Gamma \vdash^{*}_{\mathsf{v}} e : \forall a.\,\upsilon'$, where $\overline{b} = \mathit{ftv}(\forall a.\,\upsilon') \setminus \mathit{ftv}(\Gamma)$ and $\upsilon'[\overline{\tau}/\overline{b}] \leq_{\mathsf{hmv}} \upsilon$. Applying V_TAPP gives us $\Gamma \vdash^{*}_{\mathsf{v}} e\ @\tau : \upsilon'[\tau/a]$, and V_GEN gives us $\Gamma \vdash^{gen}_{\mathsf{v}} e\ @\tau : \forall\{\overline{c}\}.\,\upsilon'[\tau/a]$ where $\overline{c} = \mathit{ftv}(\upsilon'[\tau/a]) \setminus \mathit{ftv}(\Gamma)$. We want to show that $\forall\{\overline{c}\}.\,\upsilon'[\tau/a] \leq_{\mathsf{hmv}} \upsilon[\tau/a]$, which follows when there is some $\overline{\tau}'$, such that $\upsilon'[\tau/a][\overline{\tau}'/\overline{c}] \leq_{\mathsf{hmv}} \upsilon[\tau/a]$.

This is equivalent to exchanging the substitution, i.e. finding a $\overline{\tau}'$ such that $\upsilon'[\overline{\tau}'/\overline{c}][\tau/a] \leq_{\mathsf{hmv}} \upsilon[\tau/a]$.

By Substitution (Lemma 13), we have $\upsilon'[\overline{\tau}/\overline{b}][\tau/a] \leq_{\mathsf{hmv}} \upsilon[\tau/a]$. We also know that the $\overline{b}$ are a subset of the $\overline{c}$. So we can choose $\overline{\tau}'$ to be $\overline{\tau}$ for the $\overline{b}$, and the remaining $\overline{c}$ elsewhere, and we are done.

Case HMV_LET:

$$\frac{\Gamma \vdash_{\mathsf{hmv}} e_1 : \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{hmv}} e_2 : \sigma_2}{\Gamma \vdash_{\mathsf{hmv}} \textsf{let}\, x = e_1 \,\textsf{in}\, e_2 : \sigma_2} \quad \text{HMV\_LET}$$

The induction hypothesis gives us $\Gamma \vdash_{\mathsf{v}}^{gen} e_1 : \sigma_1'$ with $\sigma_1' \leq_{\mathsf{hmv}} \sigma_1$. The induction hypothesis also gives us $\Gamma, x{:}\sigma_1 \vdash_{\mathsf{v}}^{gen} e_2 : \sigma_2$ with $\sigma_2' \leq_{\mathsf{hmv}} \sigma_2$. Use Lemma 15 to get $\Gamma, x{:}\sigma_1' \vdash_{\mathsf{v}}^{gen} e_2 : \sigma_2''$ where $\sigma_2'' \leq_{\mathsf{hmv}} \sigma_2'$.

Let $\sigma_2'' = \forall\{\overline{b}\}.\, \upsilon$ where $\overline{b} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)$. Inverting $\vdash_{\mathsf{v}}^{gen}$ gives us $\Gamma, x{:}\sigma_1' \vdash_{\mathsf{v}}^{*} e_2 : \upsilon$. We then use V_LET to get $\Gamma \vdash_{\mathsf{v}}^{*} \textsf{let}\, x = e_1 \,\textsf{in}\, e_2 : \upsilon$. Generalizing gives us $\Gamma \vdash_{\mathsf{v}}^{gen} \textsf{let}\, x = e_1 \,\textsf{in}\, e_2 : \forall\{\overline{b}\}.\, \upsilon$.

Transitivity of $\leq_{\mathsf{hmv}}$ (Lemma 12) gives us $\forall\{\overline{b}\}.\, \upsilon \leq_{\mathsf{hmv}} \sigma_2$.

Case HMV_ANNOT:

$$\frac{\upsilon \text{ closed} \qquad \upsilon = \forall \overline{a}.\, \tau \qquad \Gamma \vdash_{\mathsf{hmv}} e : \tau}{\Gamma \vdash_{\mathsf{hmv}} (e : \upsilon) : \upsilon} \quad \text{HMV\_ANNOT}$$

The induction hypothesis gives us $\Gamma \vdash_{\mathsf{v}}^{gen} e : \forall\{\overline{b}\}, \overline{b}'.\, \tau'$ with $\tau'[\overline{\tau}/\overline{b}][\overline{\tau}'/\overline{b}'] = \tau$. Inverting $\vdash_{\mathsf{v}}^{gen}$ gives us $\Gamma \vdash_{\mathsf{v}}^{*} e : \forall \overline{b}'.\, \tau'$. Applying V_INSTS gives us $\Gamma \vdash_{\mathsf{v}} e : \tau$ and we can use V_ANNOT to be done.

Case HMV_INT: Trivial.

Case HMV_GEN:

$$\frac{\Gamma \vdash_{\mathsf{hmv}} e : \sigma \qquad a \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{hmv}} e : \forall\{a\}.\, \sigma} \quad \text{HMV\_GEN}$$

The induction hypothesis gives us $\Gamma \vdash_{\mathsf{v}}^{gen} e : \sigma'$ where $\sigma' \leq_{\mathsf{hmv}} \sigma$. We know $\sigma' \leq_{\mathsf{hmv}} \forall\{a\}.\, \sigma$. In other words, if $\sigma' = \forall\{\overline{b}\}.\, \upsilon_1$ and $\sigma = \forall\{\overline{c}\}.\, \upsilon_2$, we have some $\overline{\tau}$ such that $\upsilon_1[\overline{\tau}/\overline{b}] = \upsilon_2$. By the definition of $\leq_{\mathsf{hmv}}$ we can use these same $\overline{\tau}$ to show that $\sigma' \leq_{\mathsf{hmv}} \forall\{a, \overline{c}\}.\, \upsilon_2$.

Case HMV_SUB:

$$\frac{\Gamma \vdash_{\mathsf{hmv}} e : \sigma_1 \qquad \sigma_1 \leq_{\mathsf{hmv}} \sigma_2}{\Gamma \vdash_{\mathsf{hmv}} e : \sigma_2} \quad \text{HMV\_SUB}$$

The induction hypothesis gives us $\Gamma \vdash_{\mathsf{v}}^{gen} e : \sigma'$ where $\sigma' \leq_{\mathsf{hmv}} \sigma_1$. By transitivity of $\leq_{\mathsf{hmv}}$, we are done.

□

D. Algorithm $\mathcal{V}$

In this appendix, we use metavariables Q, R, and S to refer to substitutions from type variables a to monotypes τ. We apply and compose these as functions, homomorphically lifted from type variables to types.

We suppose the existence of a unification algorithm $\mathcal{U}$, that produces a substitution S, with the following properties

- If $S = \mathcal{U}_{\overline{a}}(\tau_1, \tau_2)$ then either $S(\tau_1) = S(\tau_2)$ and $\overline{a} \cap \mathit{dom}(S) = \emptyset$, or no such S exists.
- If $R(\tau_1) = R(\tau_2)$ (and $\overline{a} \cap \mathit{dom}(R) = \emptyset$) then there exists some S such that $R = S \circ \mathcal{U}_{\overline{a}}(\tau_1, \tau_2)$. In other words, unification produces the most general unifier.

With this function, we can define three mutually recursive, partial functions that infer the type of an expression in a given context. These equations are to be read top-to-bottom.

Definition 17 (Algorithm $\mathcal{V}$)**.**

(1) $\mathcal{V}(\Gamma, \lambda x.\, e) = (S_1, S_1(b) \to \tau)$ when
 $\mathcal{V}((\Gamma, x{:}b), e) = (S_1, \tau)$
 b fresh

(2) $\mathcal{V}(\Gamma, e_1\, e_2) = (S_3 \circ S_2 \circ S_1, S_3(b))$ when
 $(S_1, \tau_1) = \mathcal{V}(\Gamma, e_1)$
 $(S_2, \tau_2) = \mathcal{V}(S_1(\Gamma), e_2)$
 $S_3 = \mathcal{U}(S_2(\tau_1), \tau_2 \to b)$
 b fresh

(3) $\mathcal{V}(\Gamma, n) = (\epsilon, \mathit{Int})$

(4) $\mathcal{V}(\Gamma, e) = (S_1, \tau)$ when
 $(S_1, \forall \overline{a}.\, \tau) = \mathcal{V}^*(\Gamma, e)$
 $\overline{a}$ fresh

(5) $\mathcal{V}^*(\Gamma, x) = (\epsilon, \upsilon)$ when
 $x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma$
 $\overline{a}$ fresh

(6) $\mathcal{V}^*(\Gamma, \textsf{let}\, x = e_1 \,\textsf{in}\, e_2) = (S_2 \circ S_1, \upsilon_2)$ when
 $(S_1, \sigma_1) = \mathcal{V}^{gen}(\Gamma, e_1)$
 $(S_2, \upsilon_2) = \mathcal{V}^*((S_1(\Gamma), x{:}\sigma_1), e_2)$

(7) $\mathcal{V}^*(\Gamma, e\ @\tau) = (S_1, \upsilon_1[\tau/a])$ when
 $(S_1, \forall a.\, \upsilon_1) = \mathcal{V}^*(\Gamma, e)$
 τ closed

(8) $\mathcal{V}^*(\Gamma, (e : \upsilon)) = (S_2 \circ S_1, \upsilon)$ when
 υ closed
 $\forall \overline{a}.\, \tau = \upsilon$
 $(S_1, \tau') = \mathcal{V}(\Gamma, e)$
 $S_2 = \mathcal{U}_{\overline{a}}(\tau, \tau')$

(9) $\mathcal{V}^*(\Gamma, e) = \mathcal{V}(\Gamma, e)$

(10) $\mathcal{V}^{gen}(\Gamma, e) = (S, \forall\{\overline{a}\}.\, \upsilon)$ when
 $(S, \upsilon) = \mathcal{V}^*(\Gamma, e)$
 $\overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(S(\Gamma))$

Lemma 18 (Soundness of Algorithm $\mathcal{V}$)**.**

1. *If $\mathcal{V}(\Gamma, e) = (S, \tau)$ then $S(\Gamma) \vdash_{\mathsf{v}} e : \tau$*
2. *If $\mathcal{V}^*(\Gamma, e) = (S, \upsilon)$ then $S(\Gamma) \vdash_{\mathsf{v}}^{*} e : \upsilon$*
3. *If $\mathcal{V}^{gen}(\Gamma, e) = (S, \sigma)$ then $S(\Gamma) \vdash_{\mathsf{v}}^{gen} e : \sigma$*

Proof. By mutual induction on the structure of e. In the text of the proof, we will proceed in order of the clauses in the statement of the lemma, though technically, we should be considering the shape of e as the outer-level structure.

1. **Case** $e = \lambda x.\, e$**:** By (1), we have $\mathcal{V}(\Gamma, \lambda x.\, e) = (S_1, S_1(b) \to \tau)$, where $(S_1, \tau) = \mathcal{V}((\Gamma, x{:}b), e)$ and b is fresh. We must show $S_1(\Gamma) \vdash_{\mathsf{v}} \lambda x.\, e : S_1(b) \to \tau$. The induction hypothesis tells us $S_1(\Gamma, x{:}b) \vdash_{\mathsf{v}} e : \tau$. Rewrite this as $S_1(\Gamma), x{:}S_1(b) \vdash_{\mathsf{v}} e : \tau$. V_ABS then gives us the desired result.

 Case $e = e_1\, e_2$**:** By (2), we have $\mathcal{V}(\Gamma, e_1\, e_2) = (S_3 \circ S_2 \circ S_1, S_3(b))$, with several side conditions from the statement of $\mathcal{V}$. Let $R = S_3 \circ S_2 \circ S_1$. We must show $R(\Gamma) \vdash_{\mathsf{v}} e_1\, e_2 : S_3(b)$. The induction hypothesis gives us $S_1(\Gamma) \vdash_{\mathsf{v}} e_1 : \tau_1$

and $S_2(S_1(\Gamma)) \vdash_{\mathsf{v}} e_2 : \tau_2$. Furthermore, we know that $S_3(S_2(\tau_1)) = S_3(\tau_2 \to b)$.

By the substitution lemma (Lemma 16), we know that $R(\Gamma) \vdash_{\mathsf{v}} e_1 : S_3(S_2(\tau_1))$ and $R(\Gamma) \vdash_{\mathsf{v}} e_2 : S_3(\tau_2)$. The first of these can be rewritten to $R(\Gamma) \vdash_{\mathsf{v}} e_1 : S_3(\tau_2 \to b)$, or $R(\Gamma) \vdash_{\mathsf{v}} e_1 : S_3(\tau_2) \to S_3(b)$. We now use V_APP to get $R(\Gamma) \vdash_{\mathsf{v}} e_1\ e_2 : S_3(b)$ as desired.

Case $e = n$**:** By (3), we have $\mathcal{V}(\Gamma, n) = (\epsilon, \mathsf{Int})$. We must prove $\Gamma \vdash_{\mathsf{v}} n : \mathsf{Int}$, which we get from V_INT.

Other cases: By (4), we have $\mathcal{V}(\Gamma, e) = (S_1, \tau)$, where $(S_1, \forall\overline{a}.\, \tau) = \mathcal{V}^*(\Gamma, e)$. By the induction hypothesis, we have $S_1(\Gamma) \vdash^*_{\mathsf{v}} e : \forall\overline{a}.\, \tau$. By V_INSTS, we have $S_1(\Gamma) \vdash_{\mathsf{v}} e : \tau[\overline{\tau}/\overline{a}]$ for our choice of $\overline{\tau}$. Choose $\overline{\tau} = \overline{a}$ and we are done.

2. **Case** $e = x$**:** By (5), we know $\mathcal{V}^*(\Gamma, x) = (\epsilon, \upsilon)$ where $x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma$. We must show $\Gamma \vdash^*_{\mathsf{v}} x : \upsilon$. This is direct from V_VAR, choosing $\overline{\tau} = \overline{a}$.

Case $e = \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$**:** By (6), we know $\mathcal{V}^*(\Gamma, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2) = (S_2 \circ S_1, \upsilon_2)$ where $(S_1, \sigma_1) = \mathcal{V}^{gen}(\Gamma, e_1)$ and $(S_2, \upsilon_2) = \mathcal{V}^*((S_1(\Gamma), x{:}\sigma_1), e_2)$. We must show $S_2(S_1(\Gamma)) \vdash^*_{\mathsf{v}} \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \upsilon_2$. The induction hypothesis gives us $S_1(\Gamma) \vdash^{gen}_{\mathsf{v}} e_1 : \sigma_1$ and $S_2(S_1(\Gamma), x{:}\sigma_1) \vdash^*_{\mathsf{v}} e_2 : \upsilon_2$. Substitution on the former gives us $S_2(S_1(\Gamma)) \vdash^{gen}_{\mathsf{v}} e_1 : S_2(\sigma_1)$ and we can rewrite the latter as $S_2(S_1(\Gamma)), x{:}S_2(\sigma_1) \vdash^*_{\mathsf{v}} e_2 : \upsilon_2$. V_LET gives us our desired result.

Case $e = e_0\, @\tau$**:** By (7), we know $\mathcal{V}^*(\Gamma, e_0\, @\tau) = (S_1, \upsilon_1[\tau/b])$ where $(S_1, \forall a.\, \upsilon_1) = \mathcal{V}^*(\Gamma, e)$. We must show $S_1(\Gamma) \vdash^*_{\mathsf{v}} e_0\, @\tau : \upsilon_1[\tau/b]$. The induction hypothesis gives us $S_1(\Gamma) \vdash^*_{\mathsf{v}} e_0 : \forall a.\, \upsilon_1$, and we are done by V_TAPP.

Case $e = (e_0 : \upsilon)$**:** By (8), we know $\mathcal{V}^*(\Gamma, (e_0 : \upsilon)) = (S_2 \circ S_1, \upsilon)$ with side conditions from the statement of $\mathcal{V}$, including $\forall\overline{a}.\, \tau = \upsilon$. We must show $S_2(S_1(\Gamma)) \vdash^*_{\mathsf{v}} (e_0 : \upsilon) : \upsilon$. The induction hypothesis gives us $S_1(\Gamma) \vdash_{\mathsf{v}} e : \tau'$ and we also know $S_2(\tau) = S_2(\tau')$. Substitution (Lemma 16) gives us $S_2(S_1(\Gamma)) \vdash_{\mathsf{v}} e : S_2(\tau')$, which can be rewritten as $S_2(S_1(\Gamma)) \vdash_{\mathsf{v}} e : S_2(\tau)$. We know that υ is closed, which means that $\mathit{ftv}(\tau) \subseteq \overline{a}$. However, $\overline{a} \cap \mathit{dom}(S_2) = \emptyset$, so $S_2(\tau) = \tau$. We thus can use V_ANNOT and we are done.

Other cases: By (9), we have $\mathcal{V}^*(\Gamma, e) = (S, \tau)$ where $(S, \tau) = \mathcal{V}(\Gamma, e)$. We must show $S(\Gamma) \vdash^*_{\mathsf{v}} e : \tau$. The induction hypothesis gives us $S(\Gamma) \vdash_{\mathsf{v}} e : \tau$. We are done by V_MONO.

3. **All cases:** By (10), we know $\mathcal{V}^{gen}(\Gamma, e) = (S, \forall\{\overline{a}\}.\, \upsilon)$ where $(S, \upsilon) = \mathcal{V}^*(\Gamma, e)$ and $\overline{a} = \mathit{ftv}(\upsilon) \backslash \mathit{ftv}(S(\Gamma))$. We must show $S(\Gamma) \vdash^{gen}_{\mathsf{v}} e : \forall\{\overline{a}\}.\, \upsilon$. The induction hypothesis gives us $S(\Gamma) \vdash^*_{\mathsf{v}} e : \upsilon$, and we are done by V_GEN.

□

Lemma 19 (Substitution/generalization)**.** *If $\overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{ftv}(\Gamma)$ and $\overline{b} = \mathit{ftv}(S(\upsilon)) \setminus \mathit{ftv}(S(\Gamma))$, then $S(\forall\{\overline{a}\}.\, \upsilon) \leq_{\mathsf{hmv}} \forall\{\overline{b}\}.\, S(\upsilon)$*

Proof. We must show $S(\forall\{\overline{a}\}.\, \upsilon) \leq_{\mathsf{hmv}} \forall\{\overline{b}\}.\, S(\upsilon)$. Simplify this to $\forall\{\overline{c}\}.\, S(\upsilon[\overline{c}/\overline{a}]) \leq_{\mathsf{hmv}} \forall\{\overline{b}\}.\, S(\upsilon)$ where the $\overline{c}$ are fresh. (They are used to implement capture-avoidance.) By the definition of $\leq_{\mathsf{hmv}}$ (HMV_INSTG), this simplifies to $S(\upsilon[\overline{c}/\overline{a}])[\overline{\tau}/\overline{c}] \leq_{\mathsf{hmv}} S(\upsilon)$, for our choice of $\overline{\tau}$. Choose $\overline{\tau} = S(\overline{a})$, yielding our wanted to be $S(\upsilon[\overline{c}/\overline{a}])[S(\overline{a})/\overline{c}] \leq_{\mathsf{hmv}} S(\upsilon)$. Simplifying again yields $S(\upsilon[\overline{c}/\overline{a}][\overline{a}/\overline{c}]) \leq_{\mathsf{hmv}} S(\upsilon)$, which is the same as $S(\upsilon) \leq_{\mathsf{hmv}} S(\upsilon)$. We are done by reflexivity of $\leq_{\mathsf{hmv}}$. □

.

Lemma 20 (Completeness of Algorithm $\mathcal{V}$)**.** *For all contexts Γ and substitutions Q:*

1. *If $Q(\Gamma) \vdash_{\mathsf{v}} e : \tau$, then $\mathcal{V}(\Gamma, e) = (S, \tau')$ and there exists R such that $Q = R \circ S$ and $\tau = R(\tau')$.*
2. *If $Q(\Gamma) \vdash^*_{\mathsf{v}} e : \upsilon$, then $\mathcal{V}^*(\Gamma, e) = (S, \upsilon')$ and there exists R such that $Q = R \circ S$ and $\upsilon = R(\upsilon')$.*
3. *If $Q(\Gamma) \vdash^{gen}_{\mathsf{v}} e : \sigma$, then $\mathcal{V}^{gen}(\Gamma, e) = (S, \sigma')$ and there exists R such that $Q = R \circ S$ and $R(\sigma') \leq_{\mathsf{hmv}} \sigma$.*

In the $Q = R \circ S$ conclusions above, we ignore any differences on type variables that are conjured up as fresh during recursive calls.

Proof. We proceed by mutual induction on typing derivations. In each case, we must provide the following pieces:
(i) The result of the call to $\mathcal{V}$ (such as (S, τ'))
(ii) The substitution R
(iii) The fact that $Q = R \circ S$
(iv) The fact that, say, $\tau = R(\tau')$
These pieces will be labeled in each case.

Case V_ABS:

$$\frac{\Gamma, x{:}\tau_1 \vdash_{\mathsf{v}} e : \tau_2}{\Gamma \vdash_{\mathsf{v}} \lambda x.\, e : \tau_1 \to \tau_2}\quad \text{V\_ABS}$$

We know that $Q(\Gamma), x{:}\tau_1 \vdash_{\mathsf{v}} e : \tau_2$. Let $\Gamma' = \Gamma, x{:}b$ (where b is fresh) and $Q' = [\tau_1/b] \circ Q$. Then, we can see that $Q'(\Gamma') \vdash_{\mathsf{v}} e : \tau_2$. We thus use the induction hypothesis to get $\mathcal{V}((\Gamma, x{:}b), e) = (S, \tau_2')$ along with R' such that $Q' = R' \circ S$ and $\tau_2 = R'(\tau_2')$. We can thus see that
(i) $\mathcal{V}(\Gamma, \lambda x.\, e) = (S, S(b) \to \tau_2')$.
We have left only to provide R such that $Q = R \circ S$ and $\tau_1 \to \tau_2 = R(S(b) \to \tau_2') = R(S(b)) \to R(\tau_2')$.
(ii) Choose $R = R'$.
By functional extensionality, $Q = R' \circ S$ iff Q applied to any argument is the same as $R' \circ S$ applied to any argument. But in our supposition that b is fresh, b is outside the domain of possible arguments to Q; thus we can conclude
(iii) $Q = Q' = R' \circ S$, ignoring the fresh b.
(iv) We can also see that $\tau_1 = Q'(b)$ by definition of Q' and that $\tau_2 = R(\tau_2')$ by the use of the induction hypothesis.

Case V_APP:

$$\frac{\Gamma \vdash_{\mathsf{v}} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathsf{v}} e_2 : \tau_1}{\Gamma \vdash_{\mathsf{v}} e_1\ e_2 : \tau_2}\quad \text{V\_APP}$$

The induction hypothesis tells us that $\mathcal{V}(\Gamma, e_1) = (S_1, \tau_1')$ with R_1 such that $Q = R_1 \circ S_1$ and $\tau_1 \to \tau_2 = R_1(\tau_1')$. Recall that we are assuming $Q(\Gamma) \vdash_{\mathsf{v}} e_1\ e_2 : \tau_2$, which can now be written as $R_1(S_1(Q)) \vdash_{\mathsf{v}} e_1\ e_2 : \tau_2$. We thus know (by inversion) $R_1(S_1(Q)) \vdash_{\mathsf{v}} e_2 : \tau_1$. We then use the induction hypothesis on this fact, but choosing Q be R_1, not the Q originally used. This use of the induction hypothesis gives us $\mathcal{V}(S_1(Q), e_2) = (S_2, \tau_2')$ with R_2 such that $R_1 = R_2 \circ S_2$ and $\tau_1 = R_2(\tau_2')$. We now must find a substitution S_3' that is a unifier of $S_2(\tau_1')$ and $\tau_2' \to b$ for some fresh b. We know $R_1(\tau_1') = \tau_1 \to \tau_2$ and $R_2(\tau_2') = \tau_1$. We can rewrite the former as $R_2(S_2(\tau_1')) = \tau_1 \to \tau_2$. Choose $S_3' = [\tau_2/b] \circ R_2$. We see that $S_3'(S_2(\tau_1')) = S_3'(\tau_2' \to b)$ as required. We now know that $\mathcal{U}(S_2(\tau_1'), \tau_2' \to b)$ will succeed with the most general unifier S_3. We thus know that
(i) $\mathcal{V}(\Gamma, e_1\ e_2) = (S_3 \circ S_2 \circ S_1, S_3(b))$.
We must now find R. By the fact that S_3 is a most general unifier, we know that $S_3' = R \circ S_3$.
(ii) Choose R to be this substitution, found my the most-general-unifier property.

(iii) Putting all the facts about substitutions together, we see that $R \circ S_3 \circ S_2 \circ S_1 = Q$ as needed (ignoring the action on the fresh b).
We must finally show $\tau_2 = R(S_3(b)) = S_3'(b)$.
(iv) This comes from the definition of S_3.
We are done.

Case V_INT:

$$\frac{}{\Gamma \vdash_{\mathsf{v}} n : \mathit{Int}} \ \text{V\_INT}$$

(i) $\mathcal{V}(\Gamma, n) = (\epsilon, \mathit{Int})$.
(ii) Choose $R = Q$.
(iii) $Q = R \circ \epsilon$, quite easily.
(iv) $Q(\mathit{Int})$ sure does equal Int.

Case V_INSTS:

$$\frac{\begin{array}{c}\Gamma \vdash^{*}_{\mathsf{v}} e : \forall \overline{a}.\, \tau \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{v}} e : \tau[\overline{\tau}/\overline{a}]} \ \text{V\_INSTS}$$

The induction hypothesis gives us $\mathcal{V}^*(\Gamma, e) = (S, \forall \overline{a}.\, \tau')$ with R' such that $Q = R' \circ S$ and $R'(\forall \overline{a}.\, \tau') = \forall \overline{a}.\, \tau$. Note that we have liberally renamed bound variables here to ensure that the quantified variables $\overline{a}$ are the same in both cases. This is surely possible because the substitution R' cannot change the number of quantified variables (noting that τ' must not have any quantified variables itself). We thus know $R'(\tau') = \tau$ and that $\overline{a} \cap \mathit{dom}(R') = \emptyset$.
(i) We can see that $\mathcal{V}(\Gamma, e) = (S, \tau')$.
(ii) Choose R to be the $[\overline{\tau}/\overline{a}] \circ R'$.
(iii) We can see that $Q = R \circ S$ as required (ignoring the action on the fresh $\overline{a}$).
We have already established that $R'(\tau') = \tau$. We must show that $R(\tau') = \tau[\overline{\tau}/\overline{a}]$.
(iv) This follows from our definition of R.
We are done.

Case V_VAR:

$$\frac{x{:}\forall \{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{v}} x : \upsilon[\overline{\tau}/\overline{a}]} \ \text{V\_VAR}$$

We know $x{:}\forall\{\overline{a}\}.\, \upsilon \in Q(\Gamma)$. Thus, there exists υ' such that $x{:}\forall\{\overline{a}\}.\, \upsilon' \in \Gamma$ where $Q(\upsilon') = \upsilon$.
(i) Thus, $\mathcal{V}^*(\Gamma, x) = (\epsilon, \upsilon')$.
(ii) Choose $R = [\overline{\tau}/\overline{a}] \circ Q$.
(iii) Clearly, $Q = R \circ \epsilon$, ignoring the action on the fresh $\overline{a}$.
We must now show that $R(\upsilon') = \upsilon[\overline{\tau}/\overline{a}]$.
(iv) This is true by construction of R.

Case V_LET:

$$\frac{\Gamma \vdash^{gen}_{\mathsf{v}} e_1 : \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{v}} e_2 : \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{v}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 : \upsilon_2} \ \text{V\_LET}$$

The induction hypothesis gives us $\mathcal{V}^{gen}(\Gamma, e_1) = (S_1, \sigma_1')$ with R_1 such that $Q = R_1 \circ S_1$ and $R_1(\sigma_1') \leq_{\mathsf{hmv}} \sigma_1$. We know (from inversion) that $R_1(S_1(\Gamma)), x{:}\sigma_1 \vdash^{*}_{\mathsf{v}} e_2 : \upsilon_2$. We also see that $R_1(S_1(\Gamma), x{:}\sigma_1') \leq_{\mathsf{hmv}} R_1(S_1(\Gamma)), x{:}\sigma_1$ and thus that $R_1(S_1(\Gamma), x{:}\sigma_1') \vdash^{*}_{\mathsf{v}} e_2 : \upsilon_2$, by context generalization (Lemma 15), which preserves heights of derivations. We thus use the induction hypothesis again to get $\mathcal{V}^*((S_1(\Gamma), x{:}\sigma_1'), e_2) = (S_2, \upsilon_2')$ with R_2 such that $R_1 = R_2 \circ S_2$ and $\upsilon_2 = R_2(\upsilon_2')$.
(i) We thus have $\mathcal{V}^*(\Gamma, \textbf{let}\, x = e_1 \,\textbf{in}\, e_2) = (S_2 \circ S_1, \upsilon_2')$.
(ii) Choose $R = R_2$.
(iii) We can see that $Q = R_2 \circ S_2 \circ S_1$ as desired.
(iv) We can further see that $R_2(\upsilon_2') = \upsilon_2$ as desired.

Case V_TAPP:

$$\frac{\tau \text{ closed} \qquad \Gamma \vdash^{*}_{\mathsf{v}} e : \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{v}} e\, @\tau : \upsilon[\tau/a]} \ \text{V\_TAPP}$$

The induction hypothesis gives us $\mathcal{V}^*(\Gamma, e) = (S, \upsilon')$ with R' such that $Q = R' \circ S$ and $\forall a.\, \upsilon = R'(\upsilon')$. Because the substitution R' maps type variables only to monotypes, we know υ' must be $\forall a.\, \upsilon''$, with $R'(\upsilon'') = \upsilon$ and $a \notin \mathit{dom}(R')$.
(i) We thus know $\mathcal{V}^*(\Gamma, e\, @\tau) = (S, \upsilon''[\tau/a])$.
(ii) Choose $R = R'$.
(iii) We already know $Q = R' \circ S$.
We must show $R'(\upsilon''[\tau/a]) = \upsilon[\tau/a]$. This can be reduced to $R'(\upsilon'')[R'(\tau)/a] = \upsilon[\tau/a]$ by the fact that $a \notin \mathit{dom}(R)$. Furthermore, we know τ is closed, so we can further reduce to $R'(\upsilon'')[\tau/a] = \upsilon[\tau/a]$.
(iv) But we know $R'(\upsilon'') = \upsilon$, so we are done.

Case V_ANNOT:

$$\frac{\upsilon \text{ closed} \qquad \upsilon = \forall \overline{a}.\, \tau \qquad \Gamma \vdash_{\mathsf{v}} e : \tau}{\Gamma \vdash^{*}_{\mathsf{v}} (e : \upsilon) : \upsilon} \ \text{V\_ANNOT}$$

The induction hypothesis gives us $\mathcal{V}(\Gamma, e) = (S_1, \tau')$ with R_1 such that $Q = R_1 \circ S_1$ and $R_1(\tau') = \tau$. We must show that τ and τ' have a unifier. We can assume (by the Barendregt convention) that $\overline{a} \cap \mathit{dom}(R_1) = \emptyset$. We also know that $\mathit{ftv}(\tau) \subseteq \overline{a}$. Thus, $R_1(\tau) = \tau$ and R_1 is a unifier of τ and τ'. Let $S_2 = \mathcal{U}_{\overline{a}}(\tau, \tau')$ (which is now sure to exist).
(i) We thus have $\mathcal{V}^*(\Gamma, (e : \upsilon)) = (S_2 \circ S_1, \upsilon)$.
(ii) Choose R as determined by the fact that $R_1 = R \circ S_2$ (gained from the most-general-unifier property).
(iii) We thus see that $Q = R \circ S_2 \circ S_1$ as desired.
(iv) Furthermore, by the fact that υ is closed, we get $R(\upsilon) = \upsilon$, as desired.

Case V_MONO:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{v}} e : \tau \\ \text{no other rule matches}\end{array}}{\Gamma \vdash^{*}_{\mathsf{v}} e : \tau} \ \text{V\_MONO}$$

The induction hypothesis gives us $\mathcal{V}(\Gamma, e) = (S, \tau')$ with R such that $Q = R \circ S$ and $R(\tau') = \tau$.
(i) We see that $\mathcal{V}^*(\Gamma, e) = (S, \tau')$.
(ii) Choose R to be the one we got from the induction hypothesis.
(iii) We see that $Q = R \circ S$.
(iv) We see that $R(\tau') = \tau$.

Case V_GEN:

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

The induction hypothesis gives us $\mathcal{V}^*(\Gamma, e) = (S, \upsilon')$ with R such that $Q = R \circ S$ and $R(\upsilon') = \upsilon$. We know $\overline{a} = \mathit{ftv}(R(\upsilon')) \setminus \mathit{ftv}(R(S(\Gamma)))$, and let $\overline{a}' = \mathit{ftv}(\upsilon') \setminus \mathit{ftv}(S(\Gamma))$.
(i) We see that $\mathcal{V}^{gen}(\Gamma, e) = (S, \forall\{\overline{a}'\}.\, \upsilon')$.
(ii) Choose R as from the induction hypothesis.
(iii) We know $Q = R \circ S$.
We must show $R(\forall\{\overline{a}'\}.\, \upsilon') \leq_{\mathsf{hmv}} \forall\{\overline{a}\}.\, \upsilon$
(iv) This is direct from Lemma 19.

□

Proof of principal types for HMV (Theorem 3). By completeness of V (Theorem 5), we have σ_0' such that $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma_0'$ and $\sigma_0' \leq_{\mathsf{hmv}} \sigma$. By completeness of Algorithm $\mathcal{V}$ (Lemma 20), we have $\mathcal{V}^{gen}(\Gamma, e) = (S, \sigma_{\mathrm{p}}')$ with R such that $\epsilon = R \circ S$ and $R(\sigma_{\mathrm{p}}') \leq_{\mathsf{hmv}} \sigma_0'$. Let $\sigma_{\mathrm{p}} = R(\sigma_{\mathrm{p}}')$. By the soundness of Algorithm $\mathcal{V}$ (Lemma 18), we know that $S(\Gamma) \vdash^{gen}_{\mathsf{v}} e : \sigma_{\mathrm{p}}'$, or equivalently: $S(\Gamma) \vdash^{gen}_{\mathsf{v}} e : S(\sigma_{\mathrm{p}})$. By substitution, we can substitute through by R to get $\Gamma \vdash^{gen}_{\mathsf{v}} e : \sigma_{\mathrm{p}}$. By soundness of V (Theorem 4), we have $\Gamma \vdash_{\mathsf{hmv}} e : \sigma_{\mathrm{p}}$. Recall that $\sigma_{\mathrm{p}} \leq_{\mathsf{hmv}} \sigma_0$. But we assumed nothing about σ_0. Thus, σ_{p} is a principal type for e. □

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

$$\frac{}{\tau \leq_{\mathsf{b}} \tau}\ \text{B\_REFL}$$

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

$$\frac{\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \phi_2}{\forall \overline{a}, \overline{b}.\, \phi_1 \leq_{\mathsf{b}} \forall \overline{a}.\, \phi_2}\ \text{B\_INSTS}$$

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

Figure 13. Inner instantiation

Proof of decidability of System V (Theorem 6). All that remains is to show that Algorithm $\mathcal{V}$ terminates. All cases in Algorithm $\mathcal{V}$ except for cases (4), (9), and (10) recur on a structural component of the input. We can observe that (4) and (9) cannot infinitely recur, because one of the other cases is guaranteed to intervene. Because (10) recurs from $\mathcal{V}^{gen}(\Gamma, e)$ to $\mathcal{V}^{*}(\Gamma, e)$, it, too cannot loop. □

E. Higher-rank systems: properties of $\leq_{\mathsf{b}}$

This section concerns the properties of the first order subsumption, higher-order instantiation relation, $\sigma_1 \leq_{\mathsf{b}} \sigma_2$. For reference this relation is repeated in Figure 13.

Lemma 21 (Monotypes are already instantiated)**.** *If* $\tau_1 \leq_{\mathsf{b}} \tau_2$ *then* $\tau_1 = \tau_2$.

Proof. By inversion. □

Lemma 22 (Substitution for instantiation)**.** *If* $\sigma_1 \leq_{\mathsf{b}} \sigma_2$ *then* $S(\sigma_1) \leq_{\mathsf{b}} S(\sigma_2)$.

Lemma 23 (Reflexivity for $\leq_{\mathsf{b}}$)**.** *For all* σ, $\sigma \leq_{\mathsf{b}} \sigma$.

Lemma 24 (Transitivity for $\leq_{\mathsf{b}}$)**.** *If* $\sigma_1 \leq_{\mathsf{b}} \sigma_2$ *and* $\sigma_2 \leq_{\mathsf{b}} \sigma_3$, *then* $\sigma_1 \leq_{\mathsf{b}} \sigma_3$.

Proof. Proof is by induction on $H(\sigma_2)$, where the H function is defined as follows:

$$\begin{array}{rcl} H(\tau) & = & 1 \\ H(\upsilon_1 - > \upsilon_2) & = & 1 + max(H\upsilon_1, H\upsilon_2) \\ & & \text{when at least one of } \upsilon_1 \text{ and } \upsilon_2 \\ & & \text{is not a monotype} \\ H(\forall \overline{a}.\, \phi) & = & 1 + H(\phi) \\ H(\forall\{\overline{a}\}.\, \upsilon) & = & 1 + H(\upsilon) \end{array}$$

Note that the H function is stable under substitution (replacing variables by monotypes).

Case σ_2 is τ, a monotype In this case, by Lemma 31, we know that σ_3 must also be τ. So the result holds by assumption.

Case σ_2 is $\upsilon_{21} \to \upsilon_{22}$ By inversion, we know that σ_1 is $\forall\{\overline{a}\}, \overline{b}, \overline{c}.\, \upsilon_{11} \to \upsilon_{12}$ such that $\upsilon_{21} \leq_{\mathsf{b}} \upsilon_{11}[\overline{\tau}/\overline{a}, \overline{c}]$ and $\upsilon_{12}[\overline{\tau}/\overline{a}, \overline{c}] \leq_{\mathsf{b}} \upsilon_{12}$.
We also know that σ_3 is $\forall\{\overline{d}\}.\, \upsilon_{31} \to \upsilon_{32}$, such that $\upsilon_{31} \leq_{\mathsf{b}} \upsilon_{21}$ and $\upsilon_{22} \leq_{\mathsf{b}} \upsilon_{32}$.
By induction, we can show $\upsilon_{31} \leq_{\mathsf{b}} \upsilon_{11}[\overline{\tau}/\overline{a}, \overline{c}]$ and $\upsilon_{12}[\overline{\tau}/\overline{a}, \overline{c}] \leq_{\mathsf{b}} \upsilon_{32}$.
This lets us conclude that $\sigma_1 \leq_{\mathsf{b}} \sigma_2$.

Case σ_2 is $\forall \overline{a}.\, \phi_2$ By inversion, we know that σ_1 is $\forall\{\overline{b}\}, \overline{a}, \overline{c}.\, \phi_1$ such that $\phi_1[\overline{\tau}/\overline{b}, \overline{c}] \leq_{\mathsf{b}} \phi_2$.
We also know that σ_3 is $\forall\{\overline{d}\}, \overline{a}_1.\, \phi_3$ where $\overline{a} = \overline{a}_1, \overline{a}_2$ and $\phi_2[\overline{\tau}'/\overline{a}_2] \leq_{\mathsf{b}} \phi_3$.
By substitution, we can show that $\phi_1[\overline{\tau}/\overline{b}, \overline{c}][\overline{\tau}'/\overline{a}_2] \leq_{\mathsf{b}} \phi_2[\overline{\tau}'/\overline{a}_2]$.
By induction, we then have $\phi_1[\overline{\tau}/\overline{b}, \overline{c}][\overline{\tau}'/\overline{a}_2] \leq_{\mathsf{b}} \phi_3$. We can then derive $\sigma_1 \leq_{\mathsf{b}} \sigma_3$ to conclude.

Case σ_2 is $\forall\{\overline{a}\}.\, \upsilon_2$ By inversion, we know that σ_1 is $\forall\{\overline{b}\}.\, \upsilon_1$ where $\upsilon_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \upsilon_2$.
We also know that σ_3 is $\forall\{\overline{c}\}.\, \upsilon_3$, where $\upsilon_2[\overline{\tau}'/\overline{a}] \leq_{\mathsf{b}} \upsilon_3$.
By substitution, we can show that $\upsilon_1[\overline{\tau}/\overline{b}][\overline{\tau}'/\overline{a}] \leq_{\mathsf{b}} \upsilon_2[\overline{\tau}'/\overline{a}]$.
Rewrite this as $\upsilon_1[\overline{\tau}[\overline{\tau}'/\overline{a}]/\overline{b}] \leq_{\mathsf{b}} \upsilon_2[\overline{\tau}'/\overline{a}]$.
By induction, we have $\upsilon_1[\overline{\tau}[\overline{\tau}'/\overline{a}]/\overline{b}] \leq_{\mathsf{b}} \upsilon_3$.
Therefore we can conclude $\forall\{\overline{b}\}.\, \upsilon_1 \leq_{\mathsf{b}} \upsilon_3$.

□

F. Higher-rank systems: properties of DSK System B

This section considers the Higher-Rank type systems with deep-skolemization, described in Section 6.

F.1 Properties of Prenex conversion

Lemma 25 (Instantiation and Prenex)**.** *If* $\upsilon \leq_{\mathsf{b}} \upsilon'$ *and* $prenex(\upsilon) = \forall \overline{a}.\, \rho$ *and* $prenex(\upsilon') = \forall \overline{b}.\, \rho'$, *then* $\forall \overline{a}.\, \rho \leq_{\mathsf{b}} \forall \overline{b}.\, \rho'$ *and* $\overline{b} \subseteq \overline{a}$.

Proof. Proof is by induction on $\upsilon \leq_{\mathsf{b}} \upsilon'$.

Case B_INSTS:

$$\frac{\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \phi_2}{\forall \overline{a}, \overline{b}.\, \phi_1 \leq_{\mathsf{b}} \forall \overline{a}.\, \phi_2}\ \text{B\_INSTS}$$

Say that $prenex(\upsilon) = \forall \overline{a}, \overline{b}, \overline{c}.\, \rho_1'$ where $prenex(\phi) = \forall \overline{c}.\, \rho_1'$.
This means that $prenex(\phi_1[\overline{\tau}/\overline{b}]) = \forall \overline{c}.\, \rho_1'[\overline{\tau}/\overline{b}]$ as $\overline{\tau}$ do not contain quantifiers.
Say also that $prenex(\upsilon') = \forall \overline{a}, \overline{d}.\, \phi_2'$ where $prenex(\phi_2) = \forall \overline{d}.\, \rho_2'$.
By induction, we have that $\rho_1'[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \rho_2'$ where $\overline{d} \subseteq \overline{c}$.
Therefore we can conclude by B_INSTS that
$\forall \overline{a}.\, \rho_1' \leq_{\mathsf{b}} \forall \overline{b}.\, \rho_2'$

Case B_FUN:

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

Note that $prenex(\upsilon_1 \to \upsilon_2) = \forall \overline{a}.\, \upsilon_1 \to \rho_2$ where $\forall \overline{a}.\, \rho_2 = prenex(\upsilon_2)$ and the $\overline{a}$ are not free in υ_1. $prenex(\upsilon_3 \to \upsilon_4) = \forall \overline{b}.\, \upsilon_3 \to \rho_4$ where $\forall \overline{b}.\, \rho_4 = prenex(\upsilon_4)$. So by induction, we have that $\forall \overline{a}.\, \rho_2 \leq_{\mathsf{b}} \forall \overline{b}.\, \rho_4$ and $\overline{b} \subseteq \overline{a}$.
By inversion, we have that $\rho_2[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \rho_4$. From this we can derive $\upsilon_1 \to \rho_2[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon_3 \to \rho_4$ and then $\forall \overline{a}.\, \upsilon_1 \to \rho_2 \leq_{\mathsf{b}} \forall \overline{b}.\, \upsilon_3 \to \rho_4$.

Case B_REFL: trivial.

□

Lemma 26 (Prenex instantiates)**.** *If* $prenex(\upsilon) = \forall \overline{a}.\, \rho$ *then* $\upsilon \leq_{\mathsf{b}} \rho$.

Proof. Proof is by induction on υ. If υ is a monotype, we are done. If $\upsilon = \upsilon_1 \to \upsilon_2$, then $prenex(\upsilon) = \forall \overline{a}.\, \upsilon_1 \to \rho_2$ where

$prenex(\upsilon_2) = \forall \overline{a}.\, \rho_2$. By induction, we know that $\upsilon_2 \leq_{\mathsf{b}} \rho$, so by B_FUN, we have $\upsilon_1 \to \upsilon_2 \leq_{\mathsf{b}} \upsilon_1 \to \rho$.

If υ is a specified polytype, of the form $\forall \overline{a}.\, \phi$, then $prenex(\upsilon_2) = \forall \overline{a}, \overline{b}.\, \rho_2'$ where $prenex(\phi) = \forall \overline{b}.\, \rho_2'$. By induction, we know that $\rho_1' \leq_{\mathsf{b}} \rho_2'$. We can then show that $\forall \overline{a}, \overline{b}.\, \rho_1' \leq_{\mathsf{b}} \rho_2'$ by B_INSTS (and instantiating variables with themselves.) □

F.2 Properties of DSK subsumption

Lemma 27 (Substitution for higher-rank subsumption). *If $\upsilon_1 \leq_{\mathsf{dsk}} \upsilon_2$ then $S(\upsilon_1) \leq_{\mathsf{dsk}} S(\upsilon_2)$. If $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$ then $S(\sigma_1) \leq_{\mathsf{dsk}} S(\upsilon_2)$.*

See Vytiniotis et al. [26] Lemma 2.7.

Lemma 28 (Reflexivity for $\leq_{\mathsf{dsk}}$). *For all υ, $\upsilon \leq_{\mathsf{dsk}} \upsilon$.*

Proof. Vytiniotis et al., [26] Lemma 2.21. □

Lemma 29 (Transitivity for $\leq_{\mathsf{dsk}}$). *If $\upsilon_1 \leq_{\mathsf{dsk}} \upsilon_2$ and $\upsilon_2 \leq_{\mathsf{dsk}} \upsilon_3$, then $\upsilon_1 \leq_{\mathsf{dsk}} \upsilon_3$.*

Proof. Vytiniotis et al., [26] Lemma 2.22. □

Lemma 30 (Single skol admissible). *If $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$ then $\sigma_1 \leq_{\mathsf{dsk}} \forall c.\, \upsilon_2$ (when c are not free in σ_1).*

Proof. Proof is by induction on $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$.

Case DSK_REFL: Direct by DSK_INSTS.

Case DSK_FUN:

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

Say $prenex(\upsilon_4) = \forall \overline{b}.\, \rho$. By inversion, we know that $\upsilon_1 \to \upsilon_2 \leq_{\mathsf{dsk}} \upsilon_3 \to \rho$. We also know that $prenex(\forall c.\, \upsilon_3 \to \upsilon_4) = \forall c, \overline{b}.\, \upsilon_3 \to \rho$, so we can conclude with DSK_INST.

Case DSK_INST:

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

We note that $prenex(\forall c.\, \upsilon_2) = \forall c, \overline{b}.\, \rho_2$, so we can just apply DSK_INST.

Case SB_INST:

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

Similar reasoning to DSK_INST.

□

Lemma 31 (Monotypes are instantiations). *If $\sigma \leq_{\mathsf{dsk}} \tau$ then $\sigma \leq_{\mathsf{b}} \tau$.*

Proof. Proof is by induction on $\sigma \leq_{\mathsf{dsk}} \tau$. In each case the result holds directly by induction. □

Lemma 32 (DSK and prenex). *For all υ, we have $\upsilon \leq_{\mathsf{dsk}} prenex(\upsilon)$.*

Proof. Proof is by induction on υ. □

Lemma 33 (DSK Subsumption contains OL Subsumption). *If $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$ then $\upsilon_1 \leq_{\mathsf{dsk}} \upsilon_2$.*

Proof. Proof is by induction on the derivation.

Case OL_B_AREFL: Trivial.

Case OL_B_AFUN: By induction.

Case OL_B_AINSTS:

$$\frac{\phi_1[\overline{\tau}/\overline{a}] \leq_{\mathsf{ol}} \phi_2 \qquad \overline{b} \notin ftv(\forall \overline{a}.\, \phi_1)}{\forall \overline{a}.\, \phi_1 \leq_{\mathsf{ol}} \forall \overline{b}.\, \phi_2} \quad \text{OL\_B\_AINSTS}$$

By induction we know that $\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{dsk}} \phi_2$. We want to show that $\forall \overline{a}.\, \phi_1 \leq_{\mathsf{dsk}} \forall \overline{b}.\, \phi_2$. Say $prenex(\phi_2) = \forall \overline{c}.\, \rho_2$, then by DSK_INST it suffices to show that $\phi_1[\overline{\tau}'/\overline{b}] \leq_{\mathsf{dsk}} \rho_2$. Note that as $\forall \overline{c}.\, \rho_2 \leq_{\mathsf{dsk}} \rho_2$, by transitivity, we can reduce this to showing $\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{dsk}} prenex(\phi_2)$.

We can finish, again by transitivity, by showing via Lemma 32, that $\phi_2 \leq_{\mathsf{dsk}} prenex(\phi_2)$.

□

Corollary 34 (DSK Subsumption contains Instantiation). *If $\upsilon_1 \leq_{\mathsf{b}} \upsilon_2$ then $\upsilon_1 \leq_{\mathsf{dsk}} \upsilon_2$.*

Proof. See above and Lemma 46. □

Lemma 35 (Transitivity of Higher-Rank subsumption I). *If $\sigma_1 \leq_{\mathsf{b}} \sigma_2$ and $\sigma_2 \leq_{\mathsf{dsk}} \upsilon_3$, then $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_3$.*

Proof. Follows from Lemmas 29 and 34. □

Lemma 36 (Transitivity of Higher-Rank subsumption II). *If $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$ and $\upsilon_2 \leq_{\mathsf{b}} \upsilon_3$, then $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_3$.*

Proof. Follows from Lemmas 29 and 34. □

F.3 Substitution

Although a more general substitution property is true for these systems, in this development we need only substitute for generalized type variables. Therefore, we state these lemmas is more restrictive forms.

Lemma 37 (Substitution for System B). *Assume that the domain of S is disjoint from the variables of Γ or the free type variables of e.*

1. *If $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$ then $\Gamma \vdash_{\mathsf{b}} e \Rightarrow S(\sigma)$*
2. *If $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$ then $\Gamma \vdash_{\mathsf{b}} e \Leftarrow S(\upsilon)$*

Lemma 38 (Substitution for System SB). *Assume that the domain of S is disjoint from the variables of Γ or the free type variables of e.*

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

F.4 Soundness of syntax-directed system

Because of the differences between the syntax-directed and the declarative system, we need the following lemma about System B to prove the soundness of System SB.

Lemma 39 (Prenex System B). *If $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \rho$ and $prenex(\phi) = \forall \overline{a}.\, \rho$ then $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \phi$.*

Proof. **Case B_DABS**

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2} \quad \text{B\_DABS}$$

In this case, because we are pushing in ρ, we have $\upsilon_2 = \rho_2$. We also know that $prenex(\phi) = \forall \overline{a}.\, \upsilon_1 \to \rho_2$, so ϕ must be of the form $\upsilon_1 \to \upsilon_2$.

Write υ_2 as $\forall\overline{a}_1.\,\phi_2$. We also know that $prenex(\upsilon_2) = \forall\overline{a}_1,\overline{a}_2.\,\rho_2$ where $prenex(\phi_2) = \forall\overline{a}_2.\,\rho_2$.
By induction, we have $\Gamma, x{:}\upsilon_1 \vdash_{\overline{\mathsf{b}}} e \Leftarrow \phi_2$ By (repeated) use of B_SKOL, we have $\Gamma, x{:}\upsilon_1 \vdash_{\overline{\mathsf{b}}} e \Leftarrow \forall\overline{a}.\,\phi_2$ By B_DABS, we can then conclude $\Gamma \vdash_{\overline{\mathsf{b}}} \lambda x.\,e \Leftarrow \upsilon_1 \to \forall\overline{a}.\,\phi_2$, which is what we wanted to show.

Case B_DLET

$$\frac{\Gamma \vdash_{\overline{\mathsf{b}}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\overline{\mathsf{b}}} e_2 \Leftarrow \upsilon}{\Gamma \vdash_{\overline{\mathsf{b}}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \upsilon}\ \text{B\_DLET}$$

This case holds by induction.

Case B_SKOL This case is impossible, as the conclusion does not have the form ϕ.

Case B_INFER

$$\frac{\Gamma \vdash_{\overline{\mathsf{b}}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{dsk}} \upsilon_2}{\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \upsilon_2}\ \text{B\_INFER}$$

Here we know that υ_2 is ρ. By inversion of $\sigma \leq_{\mathsf{dsk}} \rho$, we have $\sigma = \forall\{\overline{a}_1\},\overline{b}_1.\,\phi_2$ where where $prenex(\phi_2) = \forall\overline{c}.\,\rho$ and $\rho_2[\overline{\tau}/\overline{a}_1][\overline{\tau}'/\overline{b}_1] \leq_{\mathsf{dsk}} \rho$.
However, as we also have $prenex(\phi) = \forall\overline{a}.\,\rho$, we can use the same information to conclude, $\sigma \leq_{\mathsf{dsk}} \phi$, and then use B_INFER to show that $\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \phi$.

□

Proof Soundness of System SB Lemma 7 states:

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

Proof. Most of the cases of this lemma follow via straightforward induction. Cases SB_SPEC, and SB_VAR are similar to the cases for V_INSTS and V_VAR, so are not shown. We include selected cases below.

Case SB_ANNOT:

$$\frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall\overline{a}.\,\phi \qquad \Gamma,\overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi}{\Gamma \vdash^{*}_{\mathsf{sb}} (e : \upsilon) \Rightarrow \upsilon}\ \text{SB\_ANNOT}$$

By induction we know that $\Gamma,\overline{a} \vdash_{\overline{\mathsf{b}}} e \Leftarrow \phi$. By B_ANNOT, we can conclude $\Gamma \vdash_{\overline{\mathsf{b}}} (e : \upsilon) \Rightarrow \upsilon$.

Case SB_INFER:

$$\frac{\begin{array}{c}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{dsk}} \rho_2 \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho_2}\ \text{SB\_INFER}$$

By induction, we know that $\Gamma \vdash_{\overline{\mathsf{b}}} e \Rightarrow \upsilon_1$. We would like to show that $\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \rho_2$. This follows immediately by B_INFER.

Case SB_DEEPSKOL

$$\frac{\begin{array}{c}prenex(\upsilon) = \forall\overline{a}.\,\rho \\ \overline{a} \notin vars(\Gamma) \qquad \Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho\end{array}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}\ \text{SB\_DEEPSKOL}$$

By induction, we have $\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \rho$. Note that if $prenex(\upsilon) = \forall\overline{a}.\,\rho$ and υ is of the form $\forall\overline{a}'.\,\phi$, then, by definition $prenex(\phi) = \forall\overline{a}''.\,\rho$ and $\overline{a} = \overline{a}',\overline{a}''$. By the prenex lemma 39, we know that $\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \phi$. We can then use multiple applications of rule B_SKOL to conclude $\Gamma \vdash_{\overline{\mathsf{b}}} e \Leftarrow \forall\overline{a}'.\,\phi$.

□

F.5 Completeness of syntax-directed system

Lemma 40 (Context Generalization). *Suppose $\Gamma' \leq_{\mathsf{b}} \Gamma$*

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

Proof. Proof is by induction on derivations.

Case SB_ABS:

$$\frac{\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\,e \Rightarrow \tau \to \upsilon}\ \text{SB\_ABS}$$

By induction, we know that $\Gamma', x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon$. Therefore, $\Gamma' \vdash_{\mathsf{sb}} \lambda x.\,e \Rightarrow \tau \to \upsilon'$ and, by SB_FUN, $\tau \to \upsilon' \leq_{\mathsf{b}} \tau \to \upsilon$.

Case SB_INT:

$$\frac{}{\Gamma \vdash_{\mathsf{sb}} n \Rightarrow \mathit{Int}}\ \text{SB\_INT}$$

Trivial.

Case SB_INSTS:

$$\frac{\begin{array}{c}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall\overline{a}.\,\phi \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi[\overline{\tau}/\overline{a}]}\ \text{SB\_INSTS}$$

By induction, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ where $\upsilon' \leq_{\mathsf{b}} \forall\overline{a}.\,\phi$. By inversion, we know that υ' must be of the form $\forall\overline{a},\overline{b}.\,\phi'$ where $\phi'[\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \phi$. By SB_INSTS, we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow (\phi'[\overline{\tau}'/\overline{b}])[\overline{\tau}/\overline{a}]$. We also need to show that $(\phi'[\overline{\tau}'/\overline{b}])[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \phi[\overline{\tau}/\overline{a}]$, which follows by substitution (Lemma 22).

Case SB_VAR:

$$\frac{x{:}\forall\{\overline{a}\}.\,\upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \upsilon[\overline{\tau}/\overline{a}]}\ \text{SB\_VAR}$$

We know that $x{:}\sigma \in \Gamma$, where $\sigma \leq_{\mathsf{b}} \forall\{\overline{a}\}.\,\upsilon$. So by inversion, σ must be $\forall\{\overline{b}\}.\,\upsilon'$ such that $\upsilon'[\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon$. Therefore, by substitution lemma 22, $\upsilon'[\overline{\tau}'/\overline{b}][\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon[\overline{\tau}/\overline{a}]$. As we know that the $\overline{a}$ are not free in υ', we can rewrite the left hand side as: $\upsilon'[\overline{\tau}'[\overline{\tau}/\overline{a}]/\overline{b}]$, and choose those types in the use of SB_VAR.

Case SB_APP:

$$\frac{\Gamma \vdash_{\mathsf{sb}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash^{*}_{\mathsf{sb}} e_1\ e_2 \Rightarrow \upsilon_2}\ \text{SB\_APP}$$

By induction we have $\Gamma' \vdash_{\mathsf{sb}} e_1 \Rightarrow \phi$ such that $\phi \leq_{\mathsf{b}} \upsilon_1 \to \upsilon_2$. By inversion, this means that ϕ must be of the form $\upsilon_1' \to \upsilon_2'$ where $\upsilon_1 \leq_{\mathsf{b}} \upsilon_1'$ and $\upsilon_2' \leq_{\mathsf{b}} \upsilon_2$.
By induction, we have $\Gamma' \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1'$. So we can conclude by SB_APP.

Case SB_LET:

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{sb}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Rightarrow \upsilon_2}\ \text{SB\_LET}$$

By induction $\Gamma' \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1'$ for $\sigma_1' \leq_{\mathsf{b}} \sigma_1$. By induction (again), $\Gamma', x{:}\sigma_1' \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2'$ for $\upsilon_2' \leq_{\mathsf{b}} \upsilon_2$. So we can conclude by SB_LET.

Case SB_TAPP:

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\,\upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} e\ @\tau \Rightarrow \upsilon[\tau/a]}\ \text{SB\_TAPP}$$

By induction $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ where $\upsilon' \leq_{\mathsf{b}} \forall a.\, \upsilon$. By inversion, υ' is of the form $\forall a.\, \upsilon_1$ where $\upsilon_1 \leq_{\mathsf{b}} \upsilon$. By substitution, $\upsilon_1[\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$.

Case SB_ANNOT:

$$\frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall \overline{a}.\, \phi \qquad \Gamma, \overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi}{\Gamma \vdash^{*}_{\mathsf{sb}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{SB\_ANNOT}$$

By induction, we have $\Gamma', \overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi$, so we can use SB_ANNOT to conclude $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon$.

Case SB_PHI:

$$\frac{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi \quad \text{no other rule matches}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \phi} \quad \text{SB\_PHI}$$

Holds directly by induction.

Case SB_GEN:

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

By induction, we have $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon$. Let $\overline{b} = \mathit{ftv}(\upsilon') \setminus \mathit{vars}(\Gamma)$. We want to prove that $\forall \{\overline{b}\}.\, \upsilon' \leq_{\mathsf{b}} \forall \{\overline{a}\}.\, \upsilon$. This holds by B_INSTG, choosing $\overline{\tau} = \overline{a}$.

Case SB_DABS:

$$\frac{\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \rho_2}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \to \rho_2} \quad \text{SB\_DABS}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\upsilon_1 \to \rho_2 \leq_{\mathsf{b}} \upsilon'$ be arbitrary. By inversion, we know that υ' is $\upsilon'_1 \to \upsilon'_2$ where $\upsilon'_1 \leq_{\mathsf{b}} \upsilon_1$ and $\rho_2 \leq_{\mathsf{b}} \upsilon'_2$.
By inversion, we know that υ'_2 must be of the form ρ'_2, as $\leq_{\mathsf{b}}$ cannot introduce specified quantifiers.
By induction, we know that $\Gamma', x{:}\upsilon'_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \rho'_2$.
So we can conclude by SB_DABS.

Case SB_INFER:

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{dsk}} \rho_2 \quad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho_2} \quad \text{SB\_INFER}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\rho_2 \leq_{\mathsf{b}} \rho'$ be arbitrary. By induction, we know that $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon_1$. We want to show that $\upsilon' \leq_{\mathsf{dsk}} \rho'$. This holds by lemmas 35 and 36.

Case SB_DLET:

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{sb}} e_2 \Leftarrow \rho_2}{\Gamma \vdash_{\mathsf{sb}} \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \Leftarrow \rho_2} \quad \text{SB\_DLET}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\rho_2 \leq_{\mathsf{b}} \rho'$ be arbitrary. By induction, we have $\Gamma' \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma'_1$ for $\sigma'_1 \leq_{\mathsf{b}} \sigma_1$. By induction (again), $\Gamma', x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \rho'$. So we can conclude by SB_DLET.

Case SB_DEEPSKOL

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

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\upsilon \leq_{\mathsf{b}} \upsilon'$ be arbitrary. Let $\mathit{prenex}(\upsilon') = \forall \overline{a}'.\, \rho'$. By Lemma 26, we know that $\rho \leq_{\mathsf{b}} \rho'$.
So by induction, $\Gamma' \vdash_{\mathsf{sb}} e \Leftarrow \rho'$
Therefore, we can use SB_DEEPSKOL to conclude.

□

Proof of Completeness of System SB Lemma 8 states:

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 $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$.

Proof. Most cases of this lemma either follow by induction, or are analogous to the completeness lemma for System V.

Case B_VAR

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \sigma} \quad \text{B\_VAR}$$

Suppose σ is for the form $\forall \{\overline{a}\}.\, \upsilon$. Then we use $\overline{a}$ to instantiate the variables $\overline{a}$. We know these $\overline{a}$ are not free in Γ by the Barendregt convention. It may be the case that generalization quantifies over more variables, i.e. $\overline{a} \subseteq \overline{a}' = \mathit{ftv}(\upsilon) \setminus \mathit{vars}(\Gamma)$, leading to a more general result type. However, that is permitted by the statement of the theorem.

Case B_ABS

$$\frac{\Gamma, x{:}\tau \vdash_{\mathsf{b}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Rightarrow \tau \to \upsilon} \quad \text{B\_ABS}$$

The induction hypothesis gives us $\Gamma, x{:}\tau \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \forall \{\overline{a}\}.\, \upsilon'$ where $\forall \{\overline{a}\}.\, \upsilon' \leq_{\mathsf{b}} \upsilon$. Inverting $\vdash^{gen}_{\mathsf{sb}}$ gives us $\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ and inverting $\leq_{\mathsf{b}}$ gives us $\upsilon'[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon$. We can then substitute and use SB_ABS to get $\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \to \upsilon'[\overline{\tau}/\overline{a}]$. Generalizing, we get $\Gamma \vdash^{gen}_{\mathsf{sb}} \lambda x.\, e \Rightarrow \forall \{\overline{a}, \overline{a}'\}.\, \tau \to \upsilon'[\overline{\tau}/\overline{a}]$ where the new variables $\overline{a}'$ come from generalizing τ and the $\overline{\tau}'$. We are done because $(\tau \to \upsilon')[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \tau \to \upsilon$ and so $\forall \{\overline{a}, \overline{a}'\}.\, \tau \to \upsilon' \leq_{\mathsf{b}} \tau \to \upsilon$

Case B_APP

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash_{\mathsf{b}} e_1\, e_2 \Rightarrow \upsilon_2} \quad \text{B\_APP}$$

By induction we have $\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma$ for some $\sigma \leq_{\mathsf{b}} \upsilon_1 \to \upsilon_2$. So by inversion $\sigma = \forall \{\overline{a}\}, \overline{b}.\, \upsilon'_1 \to \upsilon'_2$, where $(\upsilon'_1 \to \upsilon'_2)[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon_1 \to \upsilon_2$ and $\upsilon_1 \leq_{\mathsf{b}} \upsilon'_1[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$ and $\upsilon'_2[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon_2$.
Also by inversion of $\vdash^{gen}_{\mathsf{sb}}$, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow \forall \overline{b}.\, \upsilon'_1 \to \upsilon'_2$ and that the $\overline{a}$ are not free in Γ or e_1.
By substitution, we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow \forall \overline{b}.\, (\upsilon'_1 \to \upsilon'_2)[\overline{\tau}/\overline{a}]$
And by SB_INSTS, we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow (\upsilon'_1 \to \upsilon'_2)[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.
By induction we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon'_1$ and by lemma 53, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon'_1[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.
So we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} e_1\, e_2 \Leftarrow \upsilon'_2[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.

Case B_LET

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

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma'_1$ with $\sigma'_1 \leq_{\mathsf{b}} \sigma_1$. The induction hypothesis also gives us $\Gamma, x{:}\sigma_1 \vdash^{gen}_{\mathsf{sb}} e_2 \Rightarrow \sigma_2$ with $\sigma'_2 \leq_{\mathsf{b}} \sigma_2$. Use Lemma 53 to get $\Gamma, x{:}\sigma'_1 \vdash^{gen}_{\mathsf{sb}} e_2 \Rightarrow \sigma''_2$ where $\sigma''_2 \leq_{\mathsf{b}} \sigma'_2$.
Let $\sigma''_2 = \forall \{\overline{b}\}.\, \upsilon$ where $\overline{b} = \mathit{ftv}(\upsilon) \setminus \mathit{vars}(\Gamma)$. Inverting $\vdash^{gen}_{\mathsf{sb}}$ gives us $\Gamma, x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon$. We then use SB_LET to get $\Gamma \vdash^{*}_{\mathsf{sb}} \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \Rightarrow \upsilon$. Generalizing gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} \textbf{let}\, x = e_1\, \textbf{in}\, e_2 \Rightarrow \forall \{\overline{b}\}.\, \upsilon$.
Transitivity of $\leq_{\mathsf{b}}$ (Lemma 24) gives us $\forall \{\overline{b}\}.\, \upsilon \leq_{\mathsf{hmv}} \sigma_2$.

Case B_INT

$$\frac{}{\Gamma \vdash_{\mathsf{b}} n \Rightarrow \mathit{Int}} \quad \text{B\_INT}$$

Trivial.

Case B_TAPP

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash_{\mathsf{b}} e\, @\tau \Rightarrow \upsilon[\tau/a]} \quad \text{B\_TAPP}$$

The induction hypothesis (after inverting $\vdash^{gen}_{\mathsf{sb}}$) gives us $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon'$, where $\overline{b} = \mathit{ftv}(\forall a.\, \upsilon') \setminus \mathit{vars}(\Gamma)$ and $\upsilon'[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}}$

υ. Applying SB_TAPP gives us $\Gamma \vdash^{*}_{\mathsf{sb}} e\,@\tau \Rightarrow \upsilon'[\tau/a]$, and SB_GEN gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e\,@\tau \Rightarrow \forall\{\overline{c}\}.\,\upsilon'[\tau/a]$ where $\overline{c} = \mathit{ftv}(\upsilon'[\tau/a]) \setminus \mathit{vars}(\Gamma)$. We want to show that $\forall\{\overline{c}\}.\,\upsilon'[\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$, which follows when there is some $\overline{\tau}'$, such that $\upsilon'[\tau/a][\overline{\tau}'/\overline{c}] \leq_{\mathsf{b}} \upsilon[\tau/a]$.
This is equivalent to exchanging the substiution, i.e. finding a $\overline{\tau}'$ such that $\upsilon'[\overline{\tau}'/\overline{c}][\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$.
By Substitution (Lemma 22), we have $\upsilon'[\overline{\tau}/\overline{b}][\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$. We also know that the $\overline{b}$ are a subset of the $\overline{c}$. So we can choose $\overline{\tau}'$ to be $\overline{\tau}$ for the $\overline{b}$, and the remaining $\overline{c}$ elsewhere, and we are done.

Case B_ANNOT

$$\frac{\Gamma \vdash \upsilon \qquad \upsilon = \forall\overline{a}.\,\phi \qquad \Gamma, \overline{a} \vdash_{\mathsf{b}} e \Leftarrow \phi}{\Gamma \vdash_{\mathsf{b}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{B\_ANNOT}$$

By induction, we have $\Gamma, \overline{a} \vdash^{*}_{\mathsf{sb}} e \Leftarrow \phi$. We can conclude by SB_ANNOT.

Case B_GEN

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma \qquad a \notin \mathit{vars}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall\{a\}.\,\sigma} \quad \text{B\_GEN}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$ where $\sigma' \leq_{\mathsf{b}} \sigma$. We know $\sigma' \leq_{\mathsf{b}} \forall\{a\}.\,\sigma$. In other words, if $\sigma' = \forall\{\overline{b}\}.\,\upsilon_1$ and $\sigma = \forall\{\overline{c}\}.\,\upsilon_2$, we have some $\overline{\tau}$ such that $\upsilon_1[\overline{\tau}/\overline{b}] = \upsilon_2$. By the definition of $\leq_{\mathsf{b}}$ we can use these same $\overline{\tau}$ to show that $\sigma' \leq_{\mathsf{b}} \forall\{a, \overline{c}\}.\,\upsilon_2$.

Case B_SUB

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{b}} \sigma_2}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_2} \quad \text{B\_SUB}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$ where $\sigma' \leq_{\mathsf{b}} \sigma_1$. By transitivity of $\leq_{\mathsf{b}}$, we are done.

Case B_DABS

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\,e \Leftarrow \upsilon_1 \to \upsilon_2} \quad \text{B\_DABS}$$

By induction, we have that $\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2$.
By inversion, we have $\mathit{prenex}(\upsilon_2) = \forall\overline{a}.\,\rho$ and $\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{sb}} e \Leftarrow \rho$.
Therefore by SB_DABS, we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} \lambda x.\,e \Leftarrow \upsilon_1 \to \rho$, and by SB_DEEPSKOL, we know $\Gamma \vdash^{*}_{\mathsf{sb}} \lambda x.\,e \Leftarrow \upsilon_1 \to \upsilon_2$.

Case B_DLET

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Leftarrow \upsilon} \quad \text{B\_DLET}$$

By induction we have $\Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon$. Also by induction, there is some $\sigma' \leq_{\mathsf{b}} \sigma_1$, such that $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$. By context generalization, we know that $\Gamma, x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon$. So we can use SB_DLET to conclude.

Case B_INFER

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{dsk}} \upsilon_2}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2} \quad \text{B\_INFER}$$

In this case, we know that $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$. We would like to show that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2$. Suppose $\mathit{prenex}(\upsilon_2) = \forall\overline{a}.\,\rho$, by rule SB_DEEPSKOL, it suffices to show $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$.
Suppose σ_1 is $\forall\{\overline{b}\}.\,\upsilon_1$. By inversion of $\sigma_1 \leq_{\mathsf{dsk}} \upsilon_2$, we know that $\upsilon_1[\overline{\tau}'/\overline{b}] \leq_{\mathsf{dsk}} \rho$.
By induction, there is some $\sigma' \leq_{\mathsf{b}} \sigma_1$, such that $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$.
By inversion of $\vdash^{gen}_{\mathsf{sb}}$ we know that σ' is of the form $\forall\{\overline{a}\}.\,\upsilon'$ and that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$.
By inversion of $\sigma' \leq_{\mathsf{b}} \sigma_1$, we know that $\upsilon'[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon_1$. By substitution, this means $\upsilon'[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon_1[\overline{\tau}'/\overline{b}]$. By the Barendregt convention, the $\overline{b}$ can appear in the $\overline{\tau}$ but not in υ'. So we can rewrite this substitution as $\upsilon'[\overline{\tau}[\overline{\tau}'/\overline{b}]/\overline{a}]$. We can then substitute (as the $\overline{a}$ were generalized) $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'[\overline{\tau}[\overline{\tau}'/\overline{b}]/\overline{a}]$.
By transitivity (Lemma 35) we know that $\upsilon'[\overline{\tau}[\overline{\tau}'/\overline{b}]/\overline{a}] \leq_{\mathsf{dsk}} \rho$.
So we can conclude $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ using SB_INFER.

Case B_SKOL

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon \qquad a \notin \mathit{vars}(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \forall a.\,\upsilon} \quad \text{B\_SKOL}$$

By induction, we have $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$. We would like to conclude $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \forall a.\,\upsilon$. By inversion, we have $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \rho$ where $\mathit{prenex}(\upsilon) = \forall\overline{a}.\,\rho$. However, this means that $\mathit{prenex}(\forall a.\,\upsilon) = \forall a, \overline{a}.\,\rho$. Therefore, we can conclude using SB_DEEPSKOL.

□

Unlike System V, we have not shown that the algorithm determined by System SB computes principal types. We believe that all of the complexities of that proof are already present in the corresponding proofs about System V and the extensive proofs for the bidirectional type system of GHC [26].

$\boxed{\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2}$

$$\frac{}{\tau \leq_{\mathsf{ol}} \tau} \quad \text{OL\_B\_AREFL}$$

$$\frac{\upsilon_3 \leq_{\mathsf{ol}} \upsilon_1 \qquad \upsilon_2 \leq_{\mathsf{ol}} \upsilon_4}{\upsilon_1 \to \upsilon_2 \leq_{\mathsf{ol}} \upsilon_3 \to \upsilon_4} \quad \text{OL\_B\_AFUN}$$

$$\frac{\phi_1[\overline{\tau}/\overline{a}] \leq_{\mathsf{ol}} \phi_2 \qquad \overline{b} \notin ftv(\forall \overline{a}.\, \phi_1)}{\forall \overline{a}.\, \phi_1 \leq_{\mathsf{ol}} \forall \overline{b}.\, \phi_2} \quad \text{OL\_B\_AINSTS}$$

Figure 14. Subsumption in the Odersky-Läufer type system

G. Higher-rank systems: OL variant

This section concerns the properties of a higher-rank type system that does not include deep skolemization and provides an alternative treatment of scoped type variables.

We present this system mainly for comparison with Dunfield/Krishnaswami [9] (see below). However, it also shows that the deep-skolemization relation is not an essential component of our type system. Instead of $\leq_{\mathsf{dsk}}$, it uses the Odersky-Läufer subsmption relation shown in Figure 14.

The only difference between this system and the DSK version of System B, is the use of this relation in the B_INFER rule and the treatment of scoped type variables in the B_ANNOT and B_SKOL rules. See Figure 15. The syntax-directed version of the system is in Figure 16. It differs from System SB in that it again uses the $\leq_{\mathsf{ol}}$ relation, introduces scoped type variables at SB_SKOL and does not do deep-skolemization in the checking rule for polytypes.

A note on scoped type variables The alternative mechanism for scoped type variables leads to some "strange" binding behavior. In particular, if y has the following type in the context

$$y : (\forall a.\, a \to a) \to \mathit{Int}$$

then the expression

$$y\,(\lambda x.\,(x : a))$$

would be well typed, even though the specification of y's type could be very far from this application. Likewise, we only introduce top-level variables into scope. In particular,

$$(\lambda y.\, \lambda x.\,(x : a) : \mathit{Int} \to \forall a.\, a \to a)$$

would be well-typed if we introduced scoped type variables in rule B_SKOL, but is rejected by system B.

G.1 Properties of OL subsumption

Lemma 41 (Substitution)**.** *If $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$ then $S(\upsilon_1) \leq_{\mathsf{ol}} S(\upsilon_2)$.*

Proof. Vytiniotis et al., Lemma 2.1 ([26]) □

Lemma 42 (Reflexivity for $\leq_{\mathsf{ol}}$)**.** *For all υ, $\upsilon \leq_{\mathsf{ol}} \upsilon$.*

Proof. Vytiniotis et al., Lemma 2.2 ([26]) □

Lemma 43 (Transitivity for $\leq_{\mathsf{ol}}$)**.** *If $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$ and $\upsilon_2 \leq_{\mathsf{ol}} \upsilon_3$, then $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_3$.*

Proof. Vytiniotis et al., Lemma 2.3 ([26]) □

Lemma 44 (Single skol admissible)**.** *If $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$ then $\upsilon_1 \leq_{\mathsf{ol}} \forall c.\, \upsilon_2$ (when c is not free in υ_1).*

Proof. Proof is by induction on $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$. □

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

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \sigma} \quad \text{OL\_B\_VAR}$$

$$\frac{\Gamma, x{:}\tau \vdash_{\mathsf{b}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Rightarrow \tau \to \upsilon} \quad \text{OL\_B\_ABS}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash_{\mathsf{b}} e_1\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_B\_APP}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Rightarrow \sigma}{\Gamma \vdash_{\mathsf{b}} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Rightarrow \sigma} \quad \text{OL\_B\_LET}$$

$$\frac{}{\Gamma \vdash_{\mathsf{b}} n \Rightarrow \mathit{Int}} \quad \text{OL\_B\_INT}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash_{\mathsf{b}} e\ @\tau \Rightarrow \upsilon[\tau/a]} \quad \text{OL\_B\_TAPP}$$

$$\frac{\Gamma \vdash \upsilon \qquad \Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{OL\_B\_ANNOT}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma \qquad a \notin vars(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall \{a\}.\, \sigma} \quad \text{OL\_B\_GEN}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{b}} \sigma_2}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_2} \quad \text{OL\_B\_SUB}$$

$\boxed{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon}$

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2} \quad \text{OL\_B\_DABS}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \Leftarrow \upsilon} \quad \text{OL\_B\_DLET}$$

$$\frac{\Gamma, a \vdash_{\mathsf{b}} e \Leftarrow \upsilon \qquad a \notin vars(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \forall a.\, \upsilon} \quad \text{OL\_B\_SKOL}$$

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{ol}} \upsilon_2}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2} \quad \text{OL\_B\_INFER}$$

Figure 15. Declarative specification of System OL-B

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi}$

$$\frac{\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \rightarrow \upsilon} \quad \text{OL\_SB\_ABS}$$

$$\frac{}{\Gamma \vdash_{\mathsf{sb}} n \Rightarrow \mathit{Int}} \quad \text{OL\_SB\_INT}$$

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall \overline{a}.\, \phi \qquad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi[\overline{\tau}/\overline{a}]} \quad \text{OL\_SB\_INSTS}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}$

$$\frac{x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \upsilon[\overline{\tau}/\overline{a}]} \quad \text{OL\_SB\_VAR}$$

$$\frac{\Gamma \vdash_{\mathsf{sb}} e_1 \Rightarrow \upsilon_1 \rightarrow \upsilon_2 \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash^{*}_{\mathsf{sb}} e_1\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_SB\_APP}$$

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{sb}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_SB\_LET}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} e\, @\tau \Rightarrow \upsilon[\tau/a]} \quad \text{OL\_SB\_TAPP}$$

$$\frac{\Gamma \vdash \upsilon \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{OL\_SB\_ANNOT}$$

$$\frac{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi \qquad \text{no other rule matches}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \phi} \quad \text{OL\_SB\_PHI}$$

$\boxed{\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma}$

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon \qquad \overline{a} = \mathit{ftv}(\upsilon) \setminus \mathit{vars}(\Gamma)}{\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \forall\{\overline{a}\}.\, \upsilon} \quad \text{OL\_SB\_GEN}$$

$\boxed{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi}$

$$\frac{\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \rightarrow \upsilon_2} \quad \text{OL\_SB\_DABS}$$

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{sb}} e_2 \Leftarrow \phi_2}{\Gamma \vdash_{\mathsf{sb}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Leftarrow \phi_2} \quad \text{OL\_SB\_DLET}$$

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{ol}} \phi_2 \qquad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi_2} \quad \text{OL\_SB\_INFER}$$

$\boxed{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}$

$$\frac{\Gamma, \overline{a} \vdash_{\mathsf{sb}} e \Leftarrow \phi \qquad a \notin \mathit{vars}(\Gamma)}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \forall \overline{a}.\, \phi} \quad \text{OL\_SB\_SKOL}$$

Figure 16. Syntax-directed specification of System OL-SB

Lemma 45 (Monotypes are instantiations)**.** *If $\upsilon \leq_{\mathsf{ol}} \tau$ then $\upsilon \leq_{\mathsf{b}} \tau$.*

Proof. Proof is by induction on $\upsilon \leq_{\mathsf{ol}} \tau$. In each case the result holds directly by induction. □

Lemma 46 (Subsumption contains instantiation)**.** *If $\upsilon_1 \leq_{\mathsf{b}} \upsilon_2$ then $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$.*

Proof. Proof is by induction.

Case B_REFL: Trivial
Case B_FUN: By induction.
Case B_INSTS:

$$\frac{\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \phi_2}{\forall \overline{a}, \overline{b}.\, \phi_1 \leq_{\mathsf{b}} \forall \overline{a}.\, \phi_2} \quad \text{B\_INSTS}$$

By induction we know that $\phi_1[\overline{\tau}/\overline{b}] \leq_{\mathsf{ol}} \phi_2$. We can rewrite this as $\phi_1[\overline{a}/\overline{a}][\overline{\tau}/\overline{b}] \leq_{\mathsf{ol}} \phi_2$, and conclude by DSK_INST.

□

Lemma 47 (Transitivity of Higher-Rank subsumption I)**.** *If $\upsilon_1 \leq_{\mathsf{b}} \upsilon_2$ and $\upsilon_2 \leq_{\mathsf{ol}} \upsilon_3$, then $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_3$.*

Proof. Follows from Lemmas 43 and 46. □

Lemma 48 (Transitivity of Higher-Rank subsumption II)**.** *If $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$ and $\upsilon_2 \leq_{\mathsf{b}} \upsilon_3$, then $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_3$.*

Proof. Follows from Lemmas 43 and 46. □

G.2 Substitution

Lemma 49 (Substitution for System B)**.** *Assume that the domain of S is disjoint from the variables of Γ or the free type variables of e.*

1. *If $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma$ then $\Gamma \vdash_{\mathsf{b}} e \Rightarrow S(\sigma)$*
2. *If $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$ then $\Gamma \vdash_{\mathsf{b}} e \Leftarrow S(\upsilon)$*

Lemma 50 (Substitution for System SB)**.** *Assume that the domain of S is disjoint from the variables of Γ or the free type variables of e.*

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

G.3 Other properties

Lemma 51 (Monotypes are uninformative)**.** *If $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \tau$ then $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \tau$.*

Proof. This follows because of the four checking rules, the first two have identical monotype versions, the third doesn't apply to monotypes and the last follows by B_SUB and the fact that τ is an instantiation of σ (lemma 45). □

G.4 Soundness of System SB

Lemma 52 (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* $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi$ *then* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \phi$.

Proof. Most of the cases of this lemma follow via straightforward induction. Cases SB_SPEC, and SB_VAR are similar to the cases for V_INSTS and V_VAR.

Case SB_INFER:

$$\frac{\begin{array}{l}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{ol}} \phi_2 \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi_2} \quad \text{OL\_SB\_INFER}$$

By induction, we know that $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \upsilon_1$. We can show that $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \phi_2$ by B_INFER.

Case SB_SKOL

$$\frac{\Gamma, \overline{a} \vdash_{\mathsf{sb}} e \Leftarrow \phi \qquad a \notin vars(\Gamma)}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \forall \overline{a}.\, \phi} \quad \text{OL\_SB\_SKOL}$$

By induction, we have $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \rho$. We can immediately use rule B_SKOL to conclude.

□

G.5 Completeness of SB

Generalizing and specializing syntax-directed derivations

Lemma 53 (Context Generalization)**.** *Suppose* $\Gamma' \leq_{\mathsf{b}} \Gamma$

1. *If* $\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi$ *then there exists* $\phi' \leq_{\mathsf{b}} \phi$ *such that* $\Gamma' \vdash_{\mathsf{sb}} e \Rightarrow \phi'$.
2. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon$ *then there exists* $\upsilon' \leq_{\mathsf{b}} \upsilon$ *such that* $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$.
3. *If* $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma$ *then there exists* $\sigma' \leq_{\mathsf{b}} \sigma$ *such that* $\Gamma' \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$.
4. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$ *and* $\upsilon \leq_{\mathsf{b}} \upsilon'$ *then* $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon'$.
5. *If* $\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi$ *and* $\phi \leq_{\mathsf{b}} \phi'$ *then* $\Gamma' \vdash_{\mathsf{sb}} e \Leftarrow \phi$.

Proof. Proof is by induction on derivations.

Case SB_ABS:

$$\frac{\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \to \upsilon} \quad \text{OL\_SB\_ABS}$$

By induction, we know that $\Gamma', x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon$. Therefore, $\Gamma' \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \to \upsilon'$ and, by SB_FUN, $\tau \to \upsilon' \leq_{\mathsf{b}} \tau \to \upsilon$.

Case SB_INT:

$$\frac{}{\Gamma \vdash_{\mathsf{sb}} n \Rightarrow \mathit{Int}} \quad \text{OL\_SB\_INT}$$

Trivial.

Case SB_INSTS:

$$\frac{\begin{array}{l}\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall \overline{a}.\, \phi \\ \text{no other rule matches}\end{array}}{\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi[\overline{\tau}/\overline{a}]} \quad \text{OL\_SB\_INSTS}$$

By induction, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ where $\upsilon' \leq_{\mathsf{b}} \forall \overline{a}.\, \phi$. By inversion, we know that υ' must be of the form $\forall \overline{a}, \overline{b}.\, \phi'$ where $\phi'[\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \phi$. By SB_INSTS, we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow (\phi'[\overline{\tau}'/\overline{b}])[\overline{\tau}/\overline{a}]$. We also need to show that $(\phi'[\overline{\tau}'/\overline{b}])[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \phi[\overline{\tau}/\overline{a}]$, which follows by substitution (Lemma 22).

Case SB_VAR:

$$\frac{x{:}\forall\{\overline{a}\}.\, \upsilon \in \Gamma}{\Gamma \vdash^{*}_{\mathsf{sb}} x \Rightarrow \upsilon[\overline{\tau}/\overline{a}]} \quad \text{OL\_SB\_VAR}$$

We know that $x{:}\sigma \in \Gamma$, where $\sigma \leq_{\mathsf{b}} \forall\{\overline{a}\}.\, \upsilon$. So by inversion, σ must be $\forall\{\overline{b}\}.\, \upsilon'$ such that $\upsilon'[\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon$. Therefore, by lemma 22, $\upsilon'[\overline{\tau}'/\overline{b}][\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon[\overline{\tau}/\overline{a}]$. As we know that the $\overline{a}$ are not free in υ', we can rewrite the left hand side as: $\upsilon'[\overline{\tau}'[\overline{\tau}/\overline{a}]/\overline{b}]$, and choose those types in the use of SB_VAR.

Case SB_APP:

$$\frac{\Gamma \vdash_{\mathsf{sb}} e_1 \Rightarrow \upsilon_1 \to \upsilon_2 \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash^{*}_{\mathsf{sb}} e_1\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_SB\_APP}$$

By induction we have $\Gamma' \vdash_{\mathsf{sb}} e_1 \Rightarrow \phi$ such that $\phi \leq_{\mathsf{b}} \upsilon_1 \to \upsilon_2$. By inversion, this means that ϕ must be of the form $\upsilon_1' \to \upsilon_2'$ where $\upsilon_1 \leq_{\mathsf{b}} \upsilon_1'$ and $\upsilon_2' \leq_{\mathsf{b}} \upsilon_2$.
By induction, we have $\Gamma' \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon_1'[\overline{\tau}/\overline{b}]$. So we can conclude by SB_APP.

Case SB_LET:

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2}{\Gamma \vdash^{*}_{\mathsf{sb}} \textsf{let}\, x = e_1\, \textsf{in}\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_SB\_LET}$$

By induction $\Gamma' \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1'$ for $\sigma_1' \leq_{\mathsf{b}} \sigma_1$. By induction (again), $\Gamma', x{:}\sigma_1' \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon_2'$ for $\upsilon_2' \leq_{\mathsf{b}} \upsilon_2$. So we can conclude by SB_LET.

Case SB_TAPP:

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} e\, @\tau \Rightarrow \upsilon[\tau/a]} \quad \text{OL\_SB\_TAPP}$$

By induction $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ where $\upsilon' \leq_{\mathsf{b}} \forall a.\, \upsilon$. By inversion, υ' is of the form $\forall a.\, \upsilon_1$ where $\upsilon_1 \leq_{\mathsf{b}} \upsilon$. By substitution, $\upsilon_1[\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$.

Case SB_ANNOT:

$$\frac{\Gamma \vdash \upsilon \qquad \Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon}{\Gamma \vdash^{*}_{\mathsf{sb}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{OL\_SB\_ANNOT}$$

Holds directly by induction.

Case SB_PHI:

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{sb}} e \Rightarrow \phi \\ \text{no other rule matches}\end{array}}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \phi} \quad \text{OL\_SB\_PHI}$$

Holds directly by induction.

Case SB_GEN:

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon \qquad \overline{a} = \mathit{ftv}(\upsilon) \setminus vars(\Gamma)}{\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \forall\{\overline{a}\}.\, \upsilon} \quad \text{OL\_SB\_GEN}$$

By induction, we have $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon$. Let $\overline{b} = \mathit{ftv}(\upsilon') \setminus vars(\Gamma)$. We want to prove that $\forall\{\overline{b}\}.\, \upsilon' \leq_{\mathsf{b}} \forall\{\overline{a}\}.\, \upsilon$. This holds by definition.

Case SB_DABS:

$$\frac{\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2} \quad \text{OL\_SB\_DABS}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\upsilon_1 \to \upsilon_2 \leq_{\mathsf{b}} \upsilon'$ be arbitrary. By inversion, we know that υ' is $\upsilon_1' \to \upsilon_2'$ where $\upsilon_1' \leq_{\mathsf{b}} \upsilon_1$ and $\upsilon_2 \leq_{\mathsf{b}} \upsilon_2'$. By induction, we know that $\Gamma', x{:}\upsilon_1' \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2'$. So we can conclude by SB_DABS.

Case SB_INFER:

$$\frac{\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{ol}} \phi_2 \quad \text{no other rule matches}}{\Gamma \vdash_{\mathsf{sb}} e \Leftarrow \phi_2} \quad \text{OL\_SB\_INFER}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\phi_2 \leq_{\mathsf{b}} \phi'$ be arbitrary. By induction, we know that $\Gamma' \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ for $\upsilon' \leq_{\mathsf{b}} \upsilon_1$. We want to show that $\upsilon' \leq_{\mathsf{ol}} \phi'$. This holds by lemmas 47 and 48.

Case SB_DLET:

$$\frac{\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{sb}} e_2 \Leftarrow \phi_2}{\Gamma \vdash_{\mathsf{sb}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Leftarrow \phi_2} \quad \text{OL\_SB\_DLET}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ and $\phi_2 \leq_{\mathsf{b}} \phi'$ be arbitrary. By induction, we have $\Gamma' \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma'_1$ for $\sigma'_1 \leq_{\mathsf{b}} \sigma_1$. By induction (again), $\Gamma', x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \phi'$. So we can conclude by SB_DLET.

Case SB_SKOL:

$$\frac{\Gamma, \overline{a} \vdash_{\mathsf{sb}} e \Leftarrow \phi \qquad a \notin vars(\Gamma)}{\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \forall \overline{a}.\, \phi} \quad \text{OL\_SB\_SKOL}$$

Let $\Gamma' \leq_{\mathsf{b}} \Gamma$ be arbitrary. Result holds by directly by induction and SB_SKOL.

□

Completeness theorem

Lemma 54 (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* $\Gamma \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$.

Proof. Most cases of this lemma either follow by induction, or are analogous to the completeness lemma for System V. We discuss the most novel cases are B_APP, B_ANNOT, plus the checking rules.

Case B_VAR

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_{\mathsf{b}} x \Rightarrow \sigma} \quad \text{OL\_B\_VAR}$$

Let σ be $\forall \{\overline{a}\}.\, \upsilon$. We can use the types $\overline{a}$ to instantiate the variables $\overline{a}$. We know these $\overline{a}$ are not free in Γ by the Barendregt convention. It may be the case that generalization quantifies over more variables, i.e. $\overline{a} \subseteq \overline{a}' = ftv(\upsilon) \setminus vars(\Gamma)$, leading to a more general result type. However, that is permitted by the statement of the theorem.

Case B_ABS

$$\frac{\Gamma, x{:}\tau \vdash_{\mathsf{b}} e \Rightarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Rightarrow \tau \rightarrow \upsilon} \quad \text{OL\_B\_ABS}$$

The induction hypothesis gives us $\Gamma, x{:}\tau \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \forall \{\overline{a}\}.\, \upsilon'$ where $\forall \{\overline{a}\}.\, \upsilon' \leq_{\mathsf{b}} \upsilon$. Inverting $\vdash^{gen}_{\mathsf{sb}}$ gives us $\Gamma, x{:}\tau \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$ and inverting $\leq_{\mathsf{b}}$ gives us $\upsilon'[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon$. We can then substitute and use SB_ABS to get $\Gamma \vdash_{\mathsf{sb}} \lambda x.\, e \Rightarrow \tau \rightarrow \upsilon'[\overline{\tau}/\overline{a}]$. Generalizing, we get $\Gamma \vdash^{gen}_{\mathsf{sb}} \lambda x.\, e \Rightarrow \forall \{\overline{a}, \overline{a}'\}.\, \tau \rightarrow \upsilon'[\overline{\tau}/\overline{a}]$ where the new variables $\overline{a}'$ come from generalizing τ and the $\overline{\tau}$. We are done because $(\tau \rightarrow \upsilon')[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \tau \rightarrow \upsilon$ and so $\forall \{\overline{a}, \overline{a}'\}.\, \tau \rightarrow \upsilon' \leq_{\mathsf{b}} \tau \rightarrow \upsilon$

Case B_APP

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \upsilon_1 \rightarrow \upsilon_2 \qquad \Gamma \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon_1}{\Gamma \vdash_{\mathsf{b}} e_1\, e_2 \Rightarrow \upsilon_2} \quad \text{OL\_B\_APP}$$

By induction we have $\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma$ for some $\sigma \leq_{\mathsf{b}} \upsilon_1 \rightarrow \upsilon_2$. So by inversion $\sigma = \forall \{\overline{a}\}, \overline{b}.\, \upsilon'_1 \rightarrow \upsilon'_2$, where $(\upsilon'_1 \rightarrow \upsilon'_2)[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon_1 \rightarrow \upsilon_2$ and $\upsilon_1 \leq_{\mathsf{b}} \upsilon'_1[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$ and $\upsilon'_2[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}] \leq_{\mathsf{b}} \upsilon_2$.

Also by inversion, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow \forall \overline{b}.\, \upsilon'_1 \rightarrow \upsilon'_2$ and that the $\overline{a}$ are not free in Γ or e_1.
By substitution, we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow \forall \overline{b}.\, (\upsilon'_1 \rightarrow \upsilon'_2)[\overline{\tau}/\overline{a}]$
And by SB_INSTS, we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_1 \Rightarrow (\upsilon'_1 \rightarrow \upsilon'_2)[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.
By induction we have $\Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon'_1$ and by lemma 53, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon'_1[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.
So we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} e_1\, e_2 \Leftarrow \upsilon'_2[\overline{\tau}/\overline{a}][\overline{\tau}'/\overline{b}]$.

Case B_LET

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Rightarrow \sigma}{\Gamma \vdash_{\mathsf{b}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \sigma} \quad \text{OL\_B\_LET}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e_1 \Rightarrow \sigma'_1$ with $\sigma'_1 \leq_{\mathsf{b}} \sigma_1$. The induction hypothesis also gives us $\Gamma, x{:}\sigma_1 \vdash^{gen}_{\mathsf{sb}} e_2 \Rightarrow \sigma_2$ with $\sigma'_2 \leq_{\mathsf{b}} \sigma_2$. Use Lemma 53 to get $\Gamma, x{:}\sigma'_1 \vdash^{gen}_{\mathsf{sb}} e_2 \Rightarrow \sigma''_2$ where $\sigma''_2 \leq_{\mathsf{b}} \sigma'_2$.
Let $\sigma''_2 = \forall \{\overline{b}\}.\, \upsilon$ where $\overline{b} = ftv(\upsilon) \setminus vars(\Gamma)$. Inverting $\vdash^{gen}_{\mathsf{sb}}$ gives us $\Gamma, x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Rightarrow \upsilon$. We then use SB_LET to get $\Gamma \vdash^{*}_{\mathsf{sb}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \upsilon$. Generalizing gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} \textbf{let}\, x = e_1 \,\textbf{in}\, e_2 \Rightarrow \forall \{\overline{b}\}.\, \upsilon$.
Transitivity of $\leq_{\mathsf{b}}$ (Lemma 24) gives us $\forall \{\overline{b}\}.\, \upsilon \leq_{\mathsf{hmv}} \sigma_2$.

Case B_INT

$$\frac{}{\Gamma \vdash_{\mathsf{b}} n \Rightarrow \mathit{Int}} \quad \text{OL\_B\_INT}$$

Trivial.

Case B_TAPP

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall a.\, \upsilon}{\Gamma \vdash_{\mathsf{b}} e\, @\tau \Rightarrow \upsilon[\tau/a]} \quad \text{OL\_B\_TAPP}$$

The induction hypothesis (after inverting $\vdash^{gen}_{\mathsf{sb}}$) gives us $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \forall a.\, \upsilon'$, where $\overline{b} = ftv(\forall a.\, \upsilon') \setminus vars(\Gamma)$ and $\upsilon'[\overline{\tau}/\overline{b}] \leq_{\mathsf{b}} \upsilon$. Applying SB_TAPP gives us $\Gamma \vdash^{*}_{\mathsf{sb}} e\, @\tau \Rightarrow \upsilon'[\tau/a]$, and SB_GEN gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e\, @\tau \Rightarrow \forall \{\overline{c}\}.\, \upsilon'[\tau/a]$ where $\overline{c} = ftv(\upsilon'[\tau/a]) \setminus vars(\Gamma)$. We want to show that $\forall \{\overline{c}\}.\, \upsilon'[\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$, which follows when there is some $\overline{\tau}'$, such that $\upsilon'[\tau/a][\overline{\tau}'/\overline{c}] \leq_{\mathsf{b}} \upsilon[\tau/a]$.
This is equivalent to exchanging the substiution, i.e. finding a $\overline{\tau}'$ such that $\upsilon'[\overline{\tau}'/\overline{c}][\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$.
By Substitution (Lemma 22), we have $\upsilon'[\overline{\tau}/\overline{b}][\tau/a] \leq_{\mathsf{b}} \upsilon[\tau/a]$. We also know that the $\overline{b}$ are a subset of the $\overline{c}$. So we can choose $\overline{\tau}'$ to be $\overline{\tau}$ for the $\overline{b}$, and the remaining $\overline{c}$ elsewhere, and we are done.

Case B_ANNOT

$$\frac{\Gamma \vdash \upsilon \qquad \Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} (e : \upsilon) \Rightarrow \upsilon} \quad \text{OL\_B\_ANNOT}$$

The induction hypothesis gives us $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$, so the result immediately follows.

Case B_GEN

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma \qquad a \notin vars(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \forall \{a\}.\, \sigma} \quad \text{OL\_B\_GEN}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$ where $\sigma' \leq_{\mathsf{b}} \sigma$. We know $\sigma' \leq_{\mathsf{b}} \forall \{a\}.\, \sigma$. In other words, if $\sigma' = \forall \{\overline{b}\}.\, \upsilon_1$ and $\sigma = \forall \{\overline{c}\}.\, \upsilon_2$, we have some $\overline{\tau}$ such that $\upsilon_1[\overline{\tau}/\overline{b}] = \upsilon_2$. By the definition of $\leq_{\mathsf{b}}$ we can use these same $\overline{\tau}$ to show that $\sigma' \leq_{\mathsf{b}} \forall \{a, \overline{c}\}.\, \upsilon_2$.

Case B_SUB

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_1 \qquad \sigma_1 \leq_{\mathsf{b}} \sigma_2}{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \sigma_2} \quad \text{OL\_B\_SUB}$$

The induction hypothesis gives us $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$ where $\sigma' \leq_{\mathsf{b}} \sigma_1$. By transitivity of $\leq_{\mathsf{b}}$, we are done.

Case B_DAbs

$$\frac{\Gamma, x{:}\upsilon_1 \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2}{\Gamma \vdash_{\mathsf{b}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2} \quad \text{OL\_B\_DABS}$$

By induction, we have that $\Gamma, x{:}\upsilon_1 \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon_2$. Therefore by SB_DAbs, we can conclude $\Gamma \vdash^{*}_{\mathsf{sb}} \lambda x.\, e \Leftarrow \upsilon_1 \to \upsilon_2$.

Case B_DLet

$$\frac{\Gamma \vdash_{\mathsf{b}} e_1 \Rightarrow \sigma_1 \qquad \Gamma, x{:}\sigma_1 \vdash_{\mathsf{b}} e_2 \Leftarrow \upsilon}{\Gamma \vdash_{\mathsf{b}} \textsf{let}\, x = e_1 \,\textsf{in}\, e_2 \Leftarrow \upsilon} \quad \text{OL\_B\_DLET}$$

By induction we have $\Gamma, x{:}\sigma_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon$. Also by induction, there is some $\sigma' \leq_{\mathsf{b}} \sigma_1$, such that $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$. By context generalization, we know that $\Gamma, x{:}\sigma'_1 \vdash^{*}_{\mathsf{sb}} e_2 \Leftarrow \upsilon$. So we can use SB_DLet to conclude.

Case B_Skol

$$\frac{\Gamma, a \vdash_{\mathsf{b}} e \Leftarrow \upsilon \qquad a \notin vars(\Gamma)}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \forall a.\, \upsilon} \quad \text{OL\_B\_SKOL}$$

By induction, we have $\Gamma, a \vdash^{*}_{\mathsf{sb}} e \Leftarrow \upsilon$. We can conclude using SB_Skol. (Actually, first inverting and then potentially applying SB_Skol multiple times).

Case B_Infer

$$\frac{\Gamma \vdash_{\mathsf{b}} e \Rightarrow \upsilon_1 \qquad \upsilon_1 \leq_{\mathsf{ol}} \upsilon_2}{\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon_2} \quad \text{OL\_B\_INFER}$$

By induction, there is some $\sigma' \leq_{\mathsf{b}} \upsilon_1$, such that $\Gamma \vdash^{gen}_{\mathsf{sb}} e \Rightarrow \sigma'$. By inversion of $\vdash^{gen}_{\mathsf{sb}}$ we know that σ' is of the form $\forall\{\overline{a}\}.\, \upsilon'$ and that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'$, where $\overline{a}$ do not appear in e and Γ.
By inversion of $\leq_{\mathsf{b}}$, we know that $\upsilon'[\overline{\tau}/\overline{a}] \leq_{\mathsf{b}} \upsilon_1$. By substitution, we know that $\Gamma \vdash^{*}_{\mathsf{sb}} e \Rightarrow \upsilon'[\overline{\tau}/\overline{a}]$.
By transitivity (Lemma 47) we know that $\upsilon'[\overline{\tau}/\overline{a}] \leq_{\mathsf{ol}} \upsilon_2$. So we can conclude using SB_Infer.

□

G.6 Comparison with Dunfield / Krishnaswami

The presence of the non-deep-skolemization System B, makes it easy for us to compare our type system with the system designed by Dunfield and Krishnaswami. (We refer to this system as DK in the following.) In particular, we can show that our system subsumes the DK system.

Suppose $\Gamma \vdash_{DK} \upsilon_1 \leqslant \upsilon_2$ is the subtyping relation from the DK paper, Figure 1.

Lemma 55 (Higher-rank subsumption contains DK subtyping)**.** *If* $\Gamma \vdash_{DK} \upsilon_1 \leqslant \upsilon_2$ *then* $\upsilon_1 \leq_{\mathsf{ol}} \upsilon_2$.

Proof. But induction on the DK subtyping judgement.

Case Decl⩽Var Immediate from DSK_Refl.
Case Decl⩽Unit Immediate from DSK_Refl.
Case Decl⩽→ Directly via induction and DSK_Fun.
Case Decl⩽ ∀L By induction and DSK_Inst.
Case Decl⩽ ∀R By induction and DSK_Inst.

□

The DK system includes an application judgement, written $\Gamma \vdash_{DK} \upsilon_1 \circ e \Rightarrow\!\!\!\Rightarrow \upsilon_2$, which means "applying a function of type υ_1 to e synthesizes type υ_2".

Our declarative system does not need this judgement because we allow *implicit instantiation* for specified polytypes. In our system, the rule B_Sub allows instantiation at any point in the judgement. However, in the DK system, instantiation is restricted to be immediately before an application or when synthesis mode and checking mode meet (via subtyping). This is what our algorithm actually does, but because we have the instantiation relation, our declarative system need not make this constraint.

Lemma 56. *If* $\Gamma \vdash_{DK} \upsilon_1 \circ e \Rightarrow\!\!\!\Rightarrow \upsilon_2$ *then there exists some* υ'_1 *such that* $\upsilon_1 \leq_{\mathsf{b}} \upsilon'_1 \to \upsilon_2$ *and* $\Gamma \vdash_{DK} e \Leftarrow \upsilon'_1$.

Proof. Proof is by induction on the judgement. It requires an observation about our instantiation judgement that if $\upsilon_1 \leq_{\mathsf{b}} \upsilon'_1 \to \upsilon_2$ and υ_1 is $\forall\overline{b}.\, \rho$, then we must have instantiated *all* of the $\overline{b}$ in the judgment. In otherwords, that $\rho[\overline{\tau}/\overline{b}] = \upsilon'_1 \to \upsilon_2$.

□

Now let $\Gamma \vdash_{DK} e \Rightarrow \upsilon$, $\Gamma \vdash_{DK} e \Leftarrow \upsilon$ be the judgements shown in Figure 4 of their paper. We can also argue that System B can typecheck the same terms.

Lemma 57.

1. *If* $\Gamma \vdash_{DK} e \Leftarrow \upsilon$ *then* $\Gamma \vdash_{\mathsf{b}} e \Leftarrow \upsilon$.
2. *If* $\Gamma \vdash_{DK} e \Rightarrow \upsilon$ *then* $\Gamma \vdash_{\mathsf{b}} e \Rightarrow \upsilon$.

Proof. Proof is by induction on derivations. Most cases have direct analogues in System B. Note that technically we replace their unit type with int. They view the checking rule for the unit value as primitive and add a synthesis rule for convenience. Our system does not have a checking rule for constants, but can derive one from the synthesis rule and B_Sub.

The only other case that requires additional reasoning is the convenience rule for inferring the types of functions. We need to use lemma 51 to convert a checking judgement for monotypes into a synthesis judgement. □