Visible Type Application (Extended version)

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

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

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

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

1 Introduction

The Hindley-Milner (HM) type system [7, 12, 18] achieves remarkable concision. While allowing a strong typing discipline, a program written in HM need not mention a single type. The brevity of HM comes at a cost, however: HM programs must not mention a single type. While this rule has long been relaxed by allowing visible type annotations (and even requiring them for various type system extensions), it remains impossible for languages based on HM, such as OCaml and Haskell, to use visible type application when calling a polymorphic function.

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 \texttt{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),

\footnote{Syntax elements appearing in a programmer’s source code are often called explicit, in contrast to implicit terms, which are inferred by the compiler. However, the implicit/explicit distinction also can indicate whether terms are computationally significant \cite{19}. Our work applies only to the inferred vs. specified distinction, so we use visible to refer to syntax elements appearing in source code.}
Declarative  |  Syntax-directed
---|---
HM  |  C  |  from Damas and Milner [8] and Clément et al. [5]
HMV |  V  |  HM types with visible type application
B  |  SB  |  Higher-rank types with visible type application

Fig. 1. The type systems studied in this paper

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

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

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

First, type instantiation cannot always be determined by unification. Some Haskell features, such as type classes [29] and GHC’s type families [3, 4, 11], do not allow the type checker to unambiguously determine type arguments from an annotation. The current workaround for this issue is the \(\text{Proxy}\) type which clutters implementations and requires careful foresight by library designers. Visible type application improves such code. (See Sect. 2.)

Second, even when type arguments can be determined from an annotation, this mechanism is not always friendly to developers. For example, the variable to instantiate could appear multiple times in the type, leading to a long annotation. Partial type signatures help [30], but they do not completely solve the problem.  

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

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

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

\(^2\) Appendix A contains an example of this issue.
a distinction between specified and generalized type quantification. The importance of this system is that it demonstrates that visible type application can be added orthogonally to the HM type system, an observation that we found obvious only in hindsight.

– System V is a syntax-directed version of HMV. This type system directly corresponds to a type inference algorithm, called \( V \). Although Algorithm \( V \) works differently than Algorithm \( W \) \[8\], it retains the ability to calculate principal types. The key insight is that we can delay the instantiation of type variables until necessary. We prove that System V is sound and complete with respect to HMV, and that Algorithm \( V \) is sound and complete with respect to System V. These results show the principal types property for HMV.

– System SB is a syntax-directed bidirectional type system with higher-rank types, i.e. higher-rank types. In extending GHC with visible type application, we were required to consider the interactions of System V with all of the many type system extensions featured in GHC. Most interactions are orthogonal, as expected from the design of V. However, GHC’s extension to support higher-rank types \[23\] changes its type inference algorithm to be bidirectional. System SB shows that our approach in designing System V straightforwardly extends to a bidirectional system. System SB’s role in this paper is twofold: to show how our approach to visible type application meshes well with type system extensions, and to be the basis for our implementation in GHC.

– System B is a novel, simple declarative specification of System SB. We prove that System SB is sound and complete with respect to System B. A similar declarative specification was not present in prior work \[23\]; this paper shows that an HM-style presentation is possible even in the case of higher-rank systems.

Our visible type application extension is part of GHC 8.0. Appendix \[B\] describes this implementation and elaborates on interactions between our algorithm and other features of GHC\[F\].

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

2 Why visible type application?

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

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

\[3\] Although the type inference algorithm in this paper uses unification to determine type instantiations, following Damas’s Algorithm \( W \) \[7\], the results in this paper are applicable to GHC’s implementation based on constraint-solving. What matters here is how constraints are generated (specified by the syntax-directed versions of type systems) not how they are solved.
However, as we shall see, programming with `Proxy` is noisy and painfully indirect. With built-in visible type application, these examples are streamlined and easier to work with. In the following example and throughout this paper, unadorned code blocks are accepted by GHC 7.10, blocks with a solid gray bar at the left are ill-typed, and blocks with a gray background are accepted only by our implementation of visible type application.

### Resolving type class ambiguity

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

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

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

```haskell
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.

```haskell
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:

---

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

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

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

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

6 Our new extension `TypeApplications` implies the extension `AllowAmbiguousTypes`, which allows our updated `normalize` definition to be accepted.
Although the `show/read` ambiguity is somewhat contrived, proxies are indeed useful in more sophisticated APIs. For example, suppose a library designer would like to allow users to choose the representation of an internal data structure to best meet the needs of their application. If the type of that data structure is not included in the input and output types of the API, then a `Proxy` argument is a way to give this control to clients.\footnote{See \url{http://stackoverflow.com/questions/27044209/haskell-why-use-proxy}}

**Other examples** More practical examples of the need for visible type application require a fair amount of build-up to motivate the need for the intricate types involved. We have included two larger examples in App. A. One builds from recent work on deferring constraints until runtime \cite{defer} and the other on translating a dependently typed program from Agda \cite{agda} into Haskell.

### 3 Our approach to visible type application

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

#### 3.1 Just what are the type parameters?

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

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

\[
\text{pid} \ (x, y) = (x, y)
\]

can be assigned any of the following most general types:

\[
\begin{align*}
(1) \forall \ a \ b. \quad & (a, b) \to (a, b) \\
(2) \forall \ a \ b. \quad & (b, a) \to (b, a) \\
(3) \forall \ c \ a \ b. \quad & (a, b) \to (a, b)
\end{align*}
\]

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

\[
\text{normalize} :: \forall \ a. \ (\text{Show} \ a, \text{Read} \ a) \Rightarrow \text{String} \to \text{String}
\]

\[
\text{normalize} \ x = \text{show} \ (\text{read} \ @a \ x)
\]

\[
\text{normalizeExpr} :: \text{String} \to \text{String}
\]

\[
\text{normalizeExpr} = \text{normalize} \ @\text{Expr}
\]
Class constraints do not have a fixed ordering in types, and it is possible that a type variable is mentioned only in a constraint. Which of the following is preferred?

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

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

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

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

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

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

Fig. 2. Why specified polytypes?

is very different depending on which “equivalent” type is chosen for `pid`:

- $$(\text{Int}, \text{Bool}) \rightarrow (\text{Int}, \text{Bool})$$ -- `pid` has type (1)
- $$(\text{Bool}, \text{Int}) \rightarrow (\text{Bool}, \text{Int})$$ -- `pid` has type (2)
- $$\forall \ b. \ (\text{Bool}, b) \rightarrow (\text{Bool}, b)$$ -- `pid` has type (3)

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

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

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

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

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

\[ \text{const} :: a \rightarrow b \rightarrow a \quad \text{-- NB: no } \forall \]

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

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

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

Our first attempt to add type application to GHC was based on our understanding of Algorithm \( W \), 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:

\[ x \@ \tau_1 \ldots \@ \tau_n \]

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

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

\[ (\lambda x \rightarrow x :: \forall a \ b. \ (a, b) \rightarrow (a, b)) \@ \text{Int} \]

Likewise, we should also allow a visible instantiation after a \texttt{let} to enable refactoring:

\[ \texttt{(let } y \texttt{=} ((\lambda x \rightarrow x) :: \forall a \ b. \ (a, b) \rightarrow (a, b)) \texttt{in } y) \texttt{ } \@ \text{Int} \]

\[ ^8 \text{ In fact, the Haskell 2010 Report } [15] \text{ defines type annotations by expanding to a } \texttt{let}-\texttt{declaration with a signature.} \]
However, how do we know that we have identified all sites where visible type applications should be allowed? Furthermore, we may have identified them all for core HM, but what happens when we go to the full language of GHC, which includes features that may expose new potential sites?

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

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

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

\[
\text{Delay instantiation of “specified” type parameters until absolutely necessary.}
\]

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

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

### 4 HM with visible type application

To make our ideas precise, we next review the declarative specification of the HM type system [7, 12, 18] (which we call System HM), and then show how to extend this specification with visible type arguments.

#### 4.1 System HM

The grammar of System HM is shown in Fig. 3. The expression language comprises the Curry-style typed \(\lambda\)-calculus with the addition of numeric literals (of
Visible Type Application (Extended version) 9

HM

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

\[
e ::= x \mid \lambda x. e \mid e_1 \ e_2
\]

expressions

\[
\tau ::= a \mid \tau_1 \rightarrow \tau_2 \mid \text{Int}
\]

monotypes

\[
\sigma ::= \forall\{\pi\}. \tau
\]

type schemes

\[
\Gamma ::= \cdot \mid \Gamma, x: \sigma
\]

contexts

This grammar extends HM:

\[
e ::= ... \mid e \at \tau \mid (\Lambda x. e : \nu)
\]

expressions

\[
\tau ::= ...
\]

monotypes

\[
v ::= \tau \mid \forall a. v
\]

spec. polytypes

\[
\sigma ::= v \mid \forall\{a\}. \sigma
\]

type schemes

\[
\Gamma ::= \cdot \mid \Gamma, x: \sigma \mid \Gamma, a
\]

contexts

We write \((e : v)\) to mean \((\Lambda x. e : \nu)\), \(\text{ftv}(\sigma)\) to be the set of type variables free in \(\sigma\), and lift \(\text{ftv}\) to contexts with \(\text{ftv}(x : \sigma, a)\) to be \(a \cup \bigcup i \text{ftv}(\sigma_i)\).

**Fig. 3.** Grammars for Systems HM and HMV

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

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

So that we can better compare this system with others in the paper, we make two changes to the standard HM rules. Neither of these changes are substantial; our version types the same programs as the original. First, in \text{HM\_Let}, we allow the type of a \text{let} expression to be a polytype \(\sigma\), instead of restricting it to be a monotype \(\tau\). We discuss this change further in Sect. 5.2. Second, we replace the usual instantiation rule with \text{HM\_Sub}. This rule allows the type of any expression to be converted to any less general type in one step (as determined by the subsumption relation \(\sigma_1 \trianglelefteq_{\text{hm}} \sigma_2\)). Note that in rule \text{HM\_InstG} the lists of variables \(\pi_1\) and \(\pi_2\) need not be the same length.

**4.2 System HMV: HM with visible types**

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

---

9 Both ways of the equivalence proof proceed by induction, liberally using instantiation, generalization, and subsumption to bridge the gap between the two systems.
Typing rules for System HM

\[ \Gamma \vdash e : \sigma \]

**HM_VAR**

\[ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \]

**HM_ABS**

\[ \frac{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \]

**HM_APP**

\[ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_2 \; e_1 : \tau_2} \]

**HM_INT**

\[ \frac{\Gamma \vdash n : \text{int}}{\Gamma \vdash n : \text{int}} \]

**HM_LET**

\[ \frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2 \quad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2} \]

**HM_GEN**

\[ \frac{\Gamma \vdash e : \forall (\alpha). \sigma \quad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall (\alpha). \sigma} \]

**HM_SUB**

\[ \frac{\sigma_1 \leq_{\text{hm}} \sigma_2}{\Gamma \vdash e : \sigma} \]

**HM_INSTG**

\[ \frac{\tau_1[\tau/\pi_1] = \tau_2 \quad \pi_2 \notin \text{ftv}(\forall (\pi_1). \tau_1)}{\forall (\pi_1). \tau_1 \leq_{\text{hm}} \forall (\pi_2). \tau_2} \]

Extra typing rules for System HMV

\[ \frac{\Gamma \vdash \tau}{\Gamma \vdash \forall a. \tau} \]

**HM_VAPP**

\[ \frac{\Gamma \vdash e : \forall \alpha. \tau \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \alpha. \tau} \]

**HM_ANNOT**

\[ \frac{\Gamma \vdash \forall \alpha. \tau \quad \pi \notin \text{ftv}(\Gamma)}{\Gamma \vdash \forall \alpha. \tau} \]

**HM_SUB**

\[ \frac{\sigma_1 \leq_{\text{hm}} \sigma_2}{\Gamma \vdash e : \forall (\alpha). \sigma} \]

**HM_INSTS**

\[ \frac{\tau_1[\tau/\beta] = \tau_2 \quad \forall \alpha, \beta, \tau_1 \leq_{\text{hm}} \forall \alpha, \tau_2}{\forall \alpha, \beta, \tau_1 \leq_{\text{hm}} \forall \alpha, \tau_2} \]

**TYSCOPED**

\[ \frac{\text{ftv}(\tau) \subseteq \Gamma}{\Gamma \vdash \tau} \]

**HM_INSTG**

\[ \frac{\forall (\pi_1). \tau_1 \leq_{\text{hm}} \forall (\pi_2). \tau_2}{v_1[\tau/\pi_1] \leq_{\text{hm}} v_2 \quad \forall (\pi_1). \tau_1 \leq_{\text{hm}} \forall (\pi_2). \tau_2} \]

**TYSCOPED**

\[ \frac{\text{ftv}(\tau) \subseteq \Gamma}{\Gamma \vdash \tau} \]

Fig. 4. Typing rules for Systems HM and HMV

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

The grammar of System HMV appears in Fig. 3. The type language is enhanced with a new intermediate form \( \forall a. \forall b. \tau \) as \( \forall a, b. \tau \). This form sits between
∀\{a, b\}. a → b ≤_{hmv} ∀\{a\}. a → a  
Works the same as ≤_{hm} for type schemes

∀a, b. a → b ≤_{hmv} Int → Int  
Can instantiate specified vars

∀a, b. a → b ≤_{hmv} ∀a, b. a → b  
Can instantiate only a tail of the specified vars

∀a, b. a → b ≤_{hmv} ∀\{b\}. Int → b  
Variables can be regeneralized

∀a, b. a → b ≤_{hmv} ∀b. Int → b  
Right-to-left nature of HMV_InstS 
forces regeneralization

∀\{a\}. a → a ≤_{hmv} ∀a. a → a  
Known vars are instantiated from the right, 
ever the left

Specified quantification is more general 
than generalized quantification

Fig. 5. Examples of HMV subsumption relation

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

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

Typing contexts Γ in HMV are enhanced with the ability to store type vari-
bles. This feature is used to implement scoped type variables, where the type 
variables π, bound in Λπ.e, are available for use in types occurring within e.

Typing rules The type system of HMV includes all of the rules of HM plus 
the new rules and relation shown at the bottom of Fig. 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 σ1 ≤_{hmv} σ2 relation, described 
down.

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

10 The grammar for System HMV redefines several metavariables. These metavariables 
then have (slightly) different meanings in different sections of this paper, but disam-
biguation should be clear from context. In analysis relating systems with different 
grammars (for example, in Lemma 1), the more restrictive grammar takes prece-
dence.
A type annotation \((\Lambda \pi.e : \nu)\), typed with HMV\_ANNOT, allows an expression to be assigned a specified polytype. We require the specified polytype to have the form \(\forall a, b.\tau\); that is, a prefix of the specified polytype’s quantified variables must be the type variables scoped in the expression. The inner expression \(e\) is then checked at type \(\tau\), with the type variables \(a\) (but not the \(b\)) in scope. Types that appear in expressions (such as in type annotations and explicit type applications) may mention only type variables that are currently in scope.

Types that appear in expressions (such as in type annotations and explicit type applications) may mention only type variables that are currently in scope. Of course, in the \(\Gamma, \pi \vdash hmv e : \tau\) premise, the variables \(\pi\) and \(\vec{b}\) may appear in \(\tau\). We call such variables skolems and say that skolemizing \(\nu\) yields \(\tau\). In effect, these variables form new type constants when type-checking \(e\). When the expression \(e\) has type \(\tau\), we know that \(e\) cannot make any assumptions about the skolems \(\pi, \vec{b}\), so we can assign \(e\) the type \(\forall a, b.\tau\). This is, in effect, specified generalization.

The relation \(\sigma_1 \leq hmv \sigma_2\) (Fig. 4) implements subsumption for System HMV. The intuition is that, if \(\sigma_1 \leq hmv \sigma_2\), then an expression of type \(\sigma_1\) can be used wherever one of type \(\sigma_2\) is expected. For type schemes, the standard notion of \(\sigma_1\) being a more general type than \(\sigma_2\) 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 \(\forall a, b, \nu_1\). The subsumption rule means that we can arbitrarily change the type of \(x\) to some \(\nu\), as long as \(\nu \leq hmv \forall a, b, \nu_1\). Therefore, \(\nu\) must be of the form \(\forall a, b, \nu_2\) so that \(x @ \tau_1 @ \tau_2\) will continue to instantiate \(a\) with \(\tau_1\) and \(b\) with \(\tau_2\). Accordingly, we cannot, say, allow subsumption to reorder specified variables.

However, it is safe to allow some instantiation of specified variables as part of subsumption, as in rule HMV\_InstS. Examine this rule closely: it instantiates variables from the right. This odd-looking design choice is critical. Continuing the example above, \(\nu\) could also be of the form \(\forall a, b, c, \nu_3\). 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 \(\leq hmv\) appear in Fig. 5.

### 4.3 Properties of System HMV

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

**Lemma 1 (Conservative Extension for HMV).** Suppose \(\Gamma\) 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, \(\Gamma \vdash_{hmv} e : \sigma\) if and only if \(\Gamma \vdash_{hm} e : \sigma\).

\(\text{Note that the Barendregt convention allows bound variables to } \alpha \text{-vary, so only the number of scoped type variables is important.}\)
This property follows directly from the definition of HMV as an extension of HM. Note, in particular, that no HM typing rule is changed in HMV and that the \( \leq_{hmv} \) relation contains \( \leq_{hm} \); 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 \( \Gamma, x : \forall \{\alpha\}. \tau \vdash_{hmv} e : \sigma \), then \( \Gamma, x : \forall \alpha. \tau \vdash_{hmv} e : \sigma \).

This property follows directly from the context generalization lemma below, noting that \( \forall \alpha. \tau \leq_{hmv} \forall \{\alpha\}. \tau \).

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

This lemma is proved in App. C.

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

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

\[
\text{let } x = ((\lambda y \rightarrow y) :: \forall a. b. (a, b) \rightarrow (a, b)) \text{ in } ...
\]

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

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

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

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

### 5 Syntax-directed versions of HM and HMV

The type systems in the previous section declare when programs are well-formed, but they are fairly far removed from an algorithm. In particular, the rules \texttt{HM\_GEN} and \texttt{HM\_SUB} can appear at any point in a typing derivation.
Typing rules for System C

\[
\begin{align*}
\Gamma \vdash x : \tau & \quad \text{C_VAR} \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2, \Gamma \vdash e_2 : \tau_1 & \quad \text{C_APP} \\
\Gamma \vdash e_1 \rightarrow \tau_2, \Gamma \vdash e_2 : \tau_1 & \quad \text{C_Abs} \\
\Gamma \vdash n : \text{Int} & \quad \text{C_INT} \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 & \quad \text{C_Let} \\
\end{align*}
\]

Generalization for System C

\[
\frac{\bar{\sigma} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)}{\Gamma \vdash e : \sigma} \quad \text{C_{GEN}}
\]

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

5.1 System C

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

However, the judgment \( \Gamma \vdash e : \tau \) is still not quite an algorithm: it makes nondeterministic guesses. For example, in the rule C_ABS, the type \( \tau_1 \) is guessed; there is no indication in the expression what the choice for \( \tau_1 \) 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 W deduces the monotypes via unification, but a constraint-based approach [25] would also work.

5.2 System V: Syntax-directed visible types

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

Like System C, System V uses multiple judgments to restrict where generalization and instantiation can occur. In particular, the system allows an expres-
Visible Type Application (Extended version) 15

\[
\Gamma \vdash e : \tau
\]

Monotype checking for System V

\[
\frac{\Gamma \vdash x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \quad \text{V\_ABS}
\]

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ e_2 : \tau_2} \quad \text{V\_APP}
\]

\[
\frac{\Gamma \vdash e : \forall \alpha. \tau}{\text{no other rule matches}} \quad \text{V\_INSTS}
\]

\[
\frac{\Gamma \vdash n : \text{Int}}{\text{V\_INT}}
\]

Specified polytype checking for System V

\[
\frac{x : \forall\{\pi\}. v \in \Gamma}{\Gamma \vdash x : v[\pi/a]} \quad \text{V\_VAR}
\]

\[
\frac{\Gamma \vdash \tau}{\Gamma \vdash e : \forall \alpha. v} \quad \text{V\_TAPP}
\]

\[
\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \upsilon_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \upsilon_2} \quad \text{V\_LET}
\]

\[
\frac{\Gamma \vdash v \quad v = \forall \pi, \beta, \tau}{\Gamma, \pi \vdash e : \tau \quad \pi, \beta \notin \text{ftv}(\Gamma)} \quad \text{V\_ANNOT}
\]

\[
\frac{\Gamma\vdash e @ \tau : v[\tau/a]}{\text{no other rule matches}} \quad \text{V\_MONO}
\]

\[
\frac{\Gamma\vdash e_1 : \tau_1 \quad \Gamma\vdash e_2 : \upsilon_2}{\Gamma \vdash (\Lambda \alpha. e_1) \ : \upsilon} \quad \text{V\_ANN}
\]

\[
\frac{\pi = \text{ftv}(v) \setminus \text{ftv}(\Gamma)}{\Gamma\vdash v : \forall\{\pi\}. v} \quad \text{V\_GEN}
\]

\[\text{Fig. 7. Typing rules for System V}\]

... tion to have a type scheme only as a result of generalization (using the judgment \(\Gamma\vdash e : \sigma\)). Generalization is, once again, available only in let-expressions.

However, the main difference that enables visible type annotation is the separation of the main typing judgment into two: \(\Gamma \vdash e : \tau\) and \(\Gamma\vdash e : \upsilon\). 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 \(\Gamma\vdash e : \upsilon\) that requires \(e\) to have a monotype, and those in \(\Gamma\vdash e : \upsilon\) that can retain specified quantification.

The first set of rules in Fig. 7, as in System C, infers a monotype for the expression. The premises of the rule V\_Abs uses this same judgment, for example, to require that the body of an abstraction have a monotype. All expressions can be assigned a monotype; if the first three rules do not apply, the last rule V\_InstS infers a polytype instead, then instantiates it to yield a monotype. Because implicit instantiation happens all at once in this rule, we do not need to worry about instantiating specified variables out of order, as we did in System HMV.
The second set of rules (the \( \vdash^* \) judgment) allows \( e \) to be assigned a specified polytype. Note that the premise of rule \( \text{V_TApp} \) uses this judgment.

Rule \( \text{V_Var} \) is like rule \( \text{C_Var} \): both look up a variable in the environment and instantiate its generalized quantified variables. The difference is that \( \text{C_Var} \)'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 \( \text{V_Var} \) rule, lazily preserving the specified ones.

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

\[
\vdash^* (\text{let } x = (\lambda y. y : \forall a. a \to a) \text{ in } x) \@ \text{Int} : \text{Int} \to \text{Int}
\]

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

Rule \( \text{V_Annot} \) is identical to rule \( \text{HMV_Annot} \). It uses the \( \vdash^* \) judgment in its premise to force instantiation of all quantified type variables before regenerating to the specified polytype \( \upsilon \). In this way, the \( \text{V_Annot} \) 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 \( \text{V_Mono} \) delegates to \( \vdash^* \) to find a monotype for the expression.

### 5.3 Relating System V to System HMV

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

**Theorem 5 (Soundness of V against HMV).**

1. If \( \Gamma \vdash e : \tau \), then \( \Gamma \vdash_{\text{hmv}} e : \tau \).
2. If \( \Gamma \vdash^* e : \upsilon \), then \( \Gamma \vdash_{\text{hmv}} e : \upsilon \).
3. If \( \Gamma \vdash^\text{gen} e : \sigma \), then \( \Gamma \vdash_{\text{hmv}} e : \sigma \).

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

The proofs of these theorems appear in App. [D].

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

It is mechanical to go from the statement of System V in Fig. [H] to an algorithm. In App. [E] we define Algorithm V which implements System V, analogous
to Algorithm $W$ which implements System C. We then prove that Algorithm $V$ is sound and complete with respect to System $V$ and that Algorithm $V$ finds principal types. Linking the pieces together gives us the proof of the principal types property for System HMV (Theorem 4). Furthermore, Algorithm $V$ is guaranteed to terminate, yielding this theorem:

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

6  Higher-rank type systems

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

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

```plaintext
let foo = \f -> (f 3, f True) in foo id
```

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

```plaintext
let foo :: (\forall a. a -> a) -> (Int, Bool)
    foo = \f -> (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:

```plaintext
let foo :: (\forall a. a -> a) -> (Int -> Int, Bool)
    foo = \f -> (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:

```plaintext
pair :: \forall a. a -> \forall b. b -> (a, b)
pair = \lambda x y -> (x, y)
```

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

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

6.1 System SB: Syntax-directed Bidirectional Type Checking

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

This system shares the same expression language of Systems HMV and V, retaining visible type application and type annotations. However, types in System SB may have non-prenex quantification. The body of a specified polytype \( \nu \) is now a \( \text{phi-type} \) \( \phi \): 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 \( \nu \)s, not \( \sigma \)s. In other words, non-prenex quantification is over only specified variables, never generalized ones. As we will see, inner quantified types are introduced only by user annotation, and thus there is no way the system could produce an inner type scheme, even if the syntactic restriction were not in place.

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

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

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

**Type checking** Rule \( \text{SB\_DABS} \) is what makes the system higher-rank. The checking judgment \( \Gamma \vdash_{\text{sb}} e \Leftarrow \rho \) pushes in a rho-type, with no top-level quantification. Thus, \( \text{SB\_DABS} \) can recognize an arrow type \( \nu_1 \rightarrow \rho_2 \). Propagating this type into an expression \( \lambda x. e \), \( \text{SB\_DABS} \) uses the type \( \nu_1 \) as \( x \)'s type when
The grammar for SB extends that for System HMV (Fig. 3):

\[ e ::= \ldots \quad \text{expressions} \quad \tau ::= \ldots \quad \text{monotypes} \]
\[ \Gamma ::= \cdot | \Gamma, x: \sigma | \Gamma, a \quad \text{contexts} \]
\[ \nu ::= \phi | \forall a. \nu \quad \text{specified polytypes} \]
\[ \sigma ::= \nu | \forall \{ a \}. \sigma \quad \text{type schemes} \]

\[ \rho ::= \tau \mid \nu \]
\[ \nu_1 \rightarrow \nu_2 \quad \text{rho-types} \]
\[ \phi ::= \tau \mid \nu \]
\[ \nu_1 \rightarrow \nu_2 \quad \text{phi-types} \]

\[ \Gamma \vdash \text{sb} e \Rightarrow \phi \]

Synthesis of types without top-level quantifiers

\[ \Gamma \vdash x: \tau \Rightarrow \nu \]
[SB - VAR]
\[ \Gamma \vdash \lambda x. e \Rightarrow \tau \rightarrow \nu \]
[SB - ABS]
\[ \Gamma \vdash n \Rightarrow \text{Int} \]
[SB - INT]

Synthesis of specified polytypes

\[ \Gamma \vdash x: \forall \{ a \}. \nu \Rightarrow \nu \]
[SB - TAPP]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \nu \]
[SB - LET]

Synthesis with generalization

\[ \Gamma \vdash \text{gen } e \Rightarrow \sigma \]
[SB - GEN]

Checking against types without top-level quantifiers

\[ \Gamma \vdash e \Leftarrow \rho \]
[SB - INFER]

Checking against specified polytypes

\[ \text{prenex}(\nu) = \forall \nu', \rho \]
[SB - DeepSkol]

Fig. 8. Syntax-directed bidirectional type system
Higher-rank instantiation

$\sigma_1 \leq_b \sigma_2$

\[
\vdash \begin{array}{c}
\frac{\tau \leq_b \tau}{B_{\text{REFL}}} & \frac{\phi_1[\tau]}{B_{\text{INST}}} \\
\tau \leq_b \tau & \phi_1[\tau/\tau] \leq_b \phi_1 \\
\phi_1 & \phi_1[\tau/\tau] \leq_b \phi_1[\tau/\tau] \\
\forall \tau, b, \phi_1 \leq_b \forall \tau, \phi_2 & \frac{\phi_1[\tau/\tau]}{B_{\text{INST}}} \\
\phi_1[\tau/\tau] \leq_b \phi_1[\tau/\tau] & \forall \{\tau\}, \phi_1 \leq_b \forall \{\tau\}, \phi_2 \leq_b \forall \{\tau\}, \phi_2 \\
\phi_1[\tau/\tau] \leq_b \phi_1[\tau/\tau] & \forall \{\tau\}, \phi_1 \leq_b \forall \{\tau\}, \phi_2 \leq_b \forall \{\tau\}, \phi_2 \\
\forall \tau, \phi_1 \leq_b \forall \tau, \phi_2 & \frac{\phi_1[\tau/\tau]}{B_{\text{INST}}} \\
\phi_1[\tau/\tau] \leq_b \phi_1[\tau/\tau] & \forall \{\tau\}, \phi_1 \leq_b \forall \{\tau\}, \phi_2 \leq_b \forall \{\tau\}, \phi_2 \\
\phi_1[\tau/\tau] \leq_b \phi_1[\tau/\tau] & \forall \{\tau\}, \phi_1 \leq_b \forall \{\tau\}, \phi_2 \leq_b \forall \{\tau\}, \phi_2
\end{array}
\]

Subsumption, after deep skolemization

$\phi_1 \leq_{\text{dsk}} \rho_2$

\[
\vdash \begin{array}{c}
\frac{\tau \leq_{\text{dsk}} \tau}{D_{\text{REFL}}} & \frac{\phi_1[\tau/\tau]}{D_{\text{INST}}} \\
\tau \leq_{\text{dsk}} \tau & \phi_1[\tau/\tau] \leq_{\text{dsk}} \rho_2 \\
\phi_1[\tau/\tau] \leq_{\text{dsk}} \rho_2 & \forall \{\tau\}, \phi_1 \leq_{\text{dsk}} \forall \{\tau\}, \phi_2 \leq_{\text{dsk}} \forall \{\tau\}, \phi_2
\end{array}
\]

Deep skolemization

$\sigma_1 \leq_{\text{dsk}} \sigma_2$

\[
\vdash \begin{array}{c}
\frac{\text{prenex}(\tau) = \forall \tau, \rho_2}{\phi_1[\tau/\tau] \leq_{\text{dsk}} \rho_2} \\
\forall \{\tau\}, \phi_1 \leq_{\text{dsk}} \forall \{\tau\}, \phi_2 \leq_{\text{dsk}} \forall \{\tau\}, \phi_2
\end{array}
\]

Define $\text{prenex}(\tau) = \forall \tau, \rho_2$ as follows:

$\text{prenex}(\forall \tau, \tau) = \forall \tau, \rho_2$

$\text{prenex}(\forall \tau, \phi_1 \rightarrow \phi_2) = \forall \tau, \phi_2, \phi_1 \rightarrow \phi_2$

where $\forall \phi_1, \rho_2 = \text{prenex}(\phi_2)$

Examples:

- $\forall a. \ a \rightarrow \forall b. \ b \rightarrow b \leq_b \text{Int} \rightarrow \text{Bool} \rightarrow \text{Bool}$ Can instantiate non-top-level vars
- $\forall a. \ a \rightarrow \forall b. \ b \rightarrow b \leq_b \text{Int} \rightarrow \forall b. \ b \rightarrow b$ Not all vars must be instantiated
- $\forall a. \ a \rightarrow \forall b. \ b \rightarrow b \leq_b \forall a. \ a \rightarrow \text{Bool} \rightarrow \text{Bool}$ Can skip a top-level quantifier
- $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Bool} \leq_b (\forall a. \ a \rightarrow a) \rightarrow \text{Bool}$ Contravariant instantiation
- $\text{Int} \rightarrow \forall a. \ b. \ a \rightarrow b \leq_b \text{Int} \rightarrow \forall b. \ b \rightarrow b$ Spec. vars are inst’d from the right
- $\text{Int} \rightarrow \forall a. \ a \rightarrow a \leq_b \forall a. \ a \rightarrow a \rightarrow a$ Cannot move $\forall$ for spec. vars
- $\text{Int} \rightarrow \forall a. \ b. \ a \rightarrow b \leq_{\text{dsk}} \text{Int} \rightarrow \forall b. \ b \rightarrow b$ Spec. $\forall$ can move with $\leq_{\text{dsk}}$
- $\text{Int} \rightarrow \forall a. \ a \rightarrow a \leq_{\text{dsk}} \forall a. \ a \rightarrow a$ Spec. $\forall$ can move with $\leq_{\text{dsk}}$
- $(\text{Int} \rightarrow \forall b. \ b \rightarrow b) \rightarrow \text{Int} \leq_{\text{dsk}} (\forall a. \ b. \ a \rightarrow b \rightarrow b) \rightarrow \text{Int}$ Contravariant out-of-order inst.
- $\forall \{a\}. \ a \rightarrow a \leq_{\text{dsk}} \forall a. \ a \rightarrow a$ Same handling of spec. and gen. vars

Fig. 9. Higher-rank subsumption relations

---

Checking $e$. This is the only place in system SB where a lambda-term can abstract over a variable with a polymorphic type. Note that the synthesis rule SB\_Abs uses a monotype for the type of $x$.\textsuperscript{12}

\textsuperscript{12} Higher-rank systems can also include an “annotated abstraction” form, $\lambda x: \tau. \ e$. This form allows higher-rank types to be synthesized for lambda expressions as well as checked. However, this form is straightforward to add but is not part of GHC, which
Rule SB_INFER mediates between the checking and synthesis judgments. When no checking rule applies, we synthesize a type and then check it according to the $\leq_{dsk}$ deep skolemization relation, taken directly from previous work and shown in Fig. 9. For brevity, we do not explain the details of this relation here, instead referring readers to Peyton Jones et al. [23, Sect. 4.6] for much deeper discussion. However, we note that there is a design choice to be made here; we could have also used Odersky–Läufer’s slightly less expressive higher-rank subsumption relation [21] instead. We present the system with deep skolemization for backwards compatibility with GHC. See App. H for a discussion of this alternative.

The entry point into the type checking judgments is through the $\Gamma \vdash^*_{sb} e \leftarrow \nu$ judgment. This judgment has just one rule, SB_DEEPSKOL. The rule skolemizes all type variables appearing at the top-level and to the right of arrows. Skolemizing here is necessary to expose a rho-type to the $\Gamma \vdash^*_{sb} e \leftarrow \rho$ judgment, so that rule SB_DAbs can fire. For example, if the algorithm is checking against type $\forall a. a \rightarrow \forall b. b \rightarrow a$, it will skolemize both $a$ and $b$, pushing in the type $a \rightarrow b \rightarrow a$. As before, by stripping off the $\forall a$ and $\forall b$, those variables behave as type constants.

The interaction between rule SB_DEEPSKOL and SB_INFER is subtle. Deep skolemization is necessary in SB_DEEPSKOL because SB_INFER uses the $\Gamma \vdash^*_{sb} e \Rightarrow \nu$ synthesis judgment in its premise, instead of the $\Gamma \vdash^*_{sb} e \Rightarrow \sigma$ 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 $\sigma$-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 Fig. 10.

### 6.2 System B: Declarative specification

Figure 11 shows the typing rules of System B, a declarative system that accepts the same programs as System SB. This declarative type system itself is a novel contribution of this work. (The systems presented in related work [10, 21, 23] are more similar to SB than to B.)

Although System B is bidirectional, it is also declarative. In particular, the use of generalization (B_GEN), subsumption (B_SUB), skolemization (B_Skol), uses patterns (beyond the scope of this paper) to bind variables in abstractions. Therefore we omit the annotated abstraction form from our formalism.

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

14 Note that SB_DEEPSKOL does not bind scoped type variables. The variables $\overline{a}$ in the rule may have originally been written in a higher-rank type that is being pushed down from a function application. Though bringing $\overline{a}$ into scope would certainly be sound, it would be quite hard for a programmer to reason about. We have decided not to do this.
Assume $\Gamma = x:\forall\{a\}.\text{Int} \rightarrow a \rightarrow a$. We wish to type-check the expression $(x:\text{Int} \rightarrow \forall a. a \rightarrow a)$. Here is a valid derivation in System B:

\[
\frac{x:\forall\{a\}.\text{Int} \rightarrow a \rightarrow a \in \Gamma}{\Gamma \vdash x \Rightarrow \forall\{a\}.\text{Int} \rightarrow a \rightarrow a} \quad \frac{\forall\{a\}.\text{Int} \rightarrow a \rightarrow a \leq_{\text{dsk}} \text{Int} \rightarrow \forall a. a \rightarrow a}{\Gamma \vdash \text{Int} \Rightarrow \forall a. a \rightarrow a} \quad \frac{\text{Int} \Rightarrow \forall a. a \rightarrow a}{\Gamma \vdash x \Rightarrow \forall a. a \rightarrow a} \quad \frac{\Gamma \vdash x \Rightarrow \text{Int} \rightarrow \forall a. a \rightarrow a}{\Gamma \vdash \text{Int} \rightarrow \forall a. a \rightarrow a}
\]

Here is a valid derivation in System SB:

\[
\frac{x:\forall\{a\}.\text{Int} \rightarrow a \rightarrow a \in \Gamma}{\Gamma \vdash x \Rightarrow \forall a. a \rightarrow a} \quad \frac{\Gamma \vdash x \Rightarrow \text{Int} \rightarrow \forall a. a \rightarrow a}{\Gamma \vdash \text{Int} \Rightarrow \forall a. a \rightarrow a} \quad \frac{\text{Int} \Rightarrow \forall a. a \rightarrow a}{\Gamma \vdash x \Rightarrow \forall a. a \rightarrow a} \quad \frac{\Gamma \vdash x \Rightarrow \text{Int} \rightarrow \forall a. a \rightarrow a}{\Gamma \vdash \text{Int} \rightarrow \forall a. a \rightarrow a}
\]

Note the deep skolemization in this derivation. If we did only a shallow skolemization at the point we use $\text{SB\_DeepSkol}$, then $a$ would not be skolemized. Accordingly, it would be impossible to instantiate the type of $x$ with $a$ in the use of the $\text{SB\_Var}$ rule.

**Fig. 10.** An example of why deep skolemization in $\text{SB\_DeepSkol}$ is necessary

and mode switching ($\text{B\_Infer}$), can happen arbitrarily in a typing derivation. Understanding what expressions are well-typed does not require knowing precisely when these operations take place.

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

In contrast, rule $\text{B\_Infer}$ (in the checking judgment) uses the stronger of the two subsumption relations $\leq_{\text{dsk}}$. This rule appears at precisely the spot in the derivation where a specified type from synthesis mode meets the specified type from checking mode. The relation $\leq_{\text{dsk}}$ subsumes $\leq_b$; that is, $\sigma_1 \leq_b \sigma_2$ implies $\sigma_1 \leq_{\text{dsk}} \sigma_2$. 

Synthesis rules for System B

$$\Gamma \vdash e \Rightarrow \sigma$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma} \quad \text{VAR}$$

$$\frac{\Gamma, x : \tau \vdash e \Rightarrow v}{\Gamma \vdash \lambda x. e \Rightarrow \tau \rightarrow v} \quad \text{ABS}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \quad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash e_1 \varepsilon e_2 \Rightarrow v_1} \quad \text{APP}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \sigma} \quad \text{LET}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \sigma} \quad \text{LET}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \forall \{a\}. \sigma} \quad \text{GEN}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \sigma} \quad \text{SUB}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \forall \{a\}. \sigma} \quad \text{GEN}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \forall \{a\}. \sigma} \quad \text{SUB}$$

Checking rules for System B

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \sigma} \quad \text{DAABS}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \sigma} \quad \text{DLET}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \sigma} \quad \text{SKOL}$$

$$\frac{\Gamma \vdash e \Rightarrow \sigma}{\Gamma \vdash e \Rightarrow \sigma} \quad \text{INFER}$$

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

**Lemma 8 (Soundness of System SB).**

1. If $\Gamma \vdash_{sb} e \Rightarrow \phi$ then $\Gamma \vdash_b e \Rightarrow \phi$.
2. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_b e \Rightarrow v$.
3. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then $\Gamma \vdash_b e \Rightarrow \sigma$.
4. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_b e \Rightarrow v$.
5. If $\Gamma \vdash_{sb} e \Rightarrow \rho$ then $\Gamma \vdash_b e \Rightarrow \rho$.

**Lemma 9 (Completeness of System SB).**

1. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then $\Gamma \vdash_{sb} e \Rightarrow \sigma'$ where $\sigma' \leq_b \sigma$.
2. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_{sb} e \Rightarrow v$.

What is the role of System B? In our experience, programmers tend to prefer the syntax-directed presentation of the system because that version is more algorithmic. As a result, it can be easier to understand why a program type checks (or doesn’t) by reasoning about System SB.
However, the fact that System B is sound and complete with respect to System SB provides properties that we can use to reason about SB. The main difference between the two is that System B divides subsumption into two different relations. The weaker \( \leq_b \) can be used at any time during synthesis, but it can only instantiate (but not generalize) specified variables. The stronger \( \leq_{dbk} \) is used at only the check/synthesis boundary but can generalize and reorder specified variables.

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

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

**Lemma 10 (Context Generalization).** Suppose \( \Gamma' \leq_b \Gamma \).

1. If \( \Gamma \vdash_b e \Rightarrow \sigma \) then there exists \( \sigma' \leq_b \sigma \) such that \( \Gamma' \vdash_b e \Rightarrow \sigma' \).
2. If \( \Gamma \vdash_b e \Leftarrow \upsilon \) and \( \upsilon \leq_b \upsilon' \) then \( \Gamma' \vdash_b e \Leftarrow \upsilon' \).

Proofs of these properties appear in App. C.

6.3 Integrating visible type application with GHC

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

One pleasing synergy between visible type application and GHC concerns GHC’s recent partial type signature feature [30]. This feature allows wildcards, written with an underscore, to appear in types; GHC infers the correct replacement for the wildcard. These work well in visible type applications, allowing the user to write @_ as a visible type argument where GHC can infer the argument. For example, if \( f \) has type \( \forall a \ b. \ a \to b \to (a, b) \), then we can write \( f \ @_ \ @_ [\text{Int}] \text{True} \ [\] \) to let GHC infer that \( a \) should be \( \text{Bool} \) but to visibly instantiate \( b \) to be \( [\text{Int}] \). Getting partial type signatures to work in the new context of visible type applications required nothing more than hooking up the pieces.

7 Related work and Conclusions

**Explicit type arguments** The F# language [20] permits explicit type arguments in function applications and method invocations. These explicit arguments, typically mandatory, are used to resolve ambiguity in type-dependent operations. However, the properties of this feature have not been studied.
Implicit arguments in dependently-typed languages  Languages such as Coq [6], Agda [20], Idris [1] and Twelf [24] are not based on the HM type system, so their designs differ from Systems HMV and B. However, they do support invisible arguments. In these languages, an invisible argument is not necessarily a type; it could be any argument that can be inferred by the type checker.

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

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

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

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

Predicative, higher-rank type systems  As we have already indicated, Systems B and SB are directly inspired by GHC’s design for higher-rank types [23]. However, in this work we have redesigned the algorithm to use lazy instantiation and have made a distinction between specified polytypes and generalized polytypes. Furthermore, we have pushed the design further, providing a declarative specification for the type system.
Our work is also closely related to recent work on using a bidirectional type system for higher-rank polymorphism by Dunfield and Krishnaswami [10], called DK below. The closest relationship is between their declarative system (Fig. 4 in their paper) and our System SB (Fig. 8). The most significant difference is that the DK system never generalizes. All polymorphic types in their system are specified; functions must have a type annotation to be polymorphic. Consequently, DK uses a different algorithm for type checking than the one proposed in this work. Nevertheless, it defers instantiations of specified polymorphism, like our algorithm.

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

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

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

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


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


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 Sect. 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:

```
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, \( \text{Bool} \) at runtime, we need runtime type information. Haskell’s Typeable feature [13] 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 wozzle.

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.

**Propositional equality: \( ∼ \):** Recent standard libraries shipped with GHC contain the following datatype:

---

15 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.

16 This example requires several GHC extensions to compile: RankNTypes, TypeOperators, DataKinds, PolyKinds, ConstraintKinds, GADTs, ScopedTypeVariables, and TypeApplications.
data $a : \sim : b$ where
  Refl :: $a : \sim : a$

This datatype implements propositional equality. If you have a value $pf :: \tau_1 : \sim : \tau_2$, that is a proof that types $\tau_1$ and $\tau_2$ are equal. Pattern matching on $pf$ reveals this equality to GHC’s type-checker, which can then use it in a pattern match:

\[
\text{boolCast :: } (a : \sim : \text{Bool}) \rightarrow a \rightarrow \text{Bool} \\
\text{boolCast } pf \ b = \text{case } pf \text{ of } \text{Refl } \rightarrow \ b
\]

Runtime cast The \textit{Typeable} feature uses $\sim$ in an important function:

\[
\text{eqT :: } \text{(Typeable } a, \text{Typeable } b) \Rightarrow \text{Maybe } (a : \sim : b)
\]

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

We are now ready to assemble the pieces:

\[
\text{instance } \text{(Typeable } a, \text{Typeable } b) \Rightarrow \text{Deferrable } (a \sim b) \text{ where} \\
\text{assume } _x = \text{case } \text{eqT :: } \text{Maybe } (a : \sim : b) \text{ of} \\
\text{Just Refl } \rightarrow \ x \\
\text{Nothing } \rightarrow \text{error "type error!"}
\]

Making assumptions Suppose we are working a list type that tracks whether it has surely one element, or whether there is an unknown length.\footnote{This example is from real code – just such a list is used within GHC when keeping track of type family axioms from either open \cite{3,4} or closed \cite{11} type families.} Here are the relevant definitions:

\[
data \text{Flag} = \text{Branched} \quad -- 0 \text{ or more elements} \\
| \text{Unbranched} \quad -- \text{exactly } 1 \text{ element} \\
data \text{List } a \ (b :: \text{Flag}) = ... \\
\text{the :: List } a \ \text{Unbranched } \rightarrow a \\
\text{the } = ...
\]

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

\[
\text{unsafeThe :: } \forall \ a \ b. \ \text{Typeable } b \Rightarrow \text{List } a \ b \rightarrow a \\
\text{unsafeThe } \ell \\
= \text{assume } \text{(Proxy :: Proxy } (b \sim \text{Unbranched})) \\
\text{(the } \ell)
\]

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

```haskell
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.

A.2 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 [10], consider a stack-based compiler for a language of boolean expressions. (The entire code for this example is available in the supplementary material.)

```haskell
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 `SEexpr` and a type-level function `Eval` that allow the type system to talk about these definitions.
\texttt{eval :: Expr \rightarrow Bool}
\texttt{eval (Val \ n) = n}
\texttt{eval (Cond e0 e1 e2) = if eval e0 then eval e1 else eval e2}

\textbf{type family Eval (x :: Expr) :: Bool where}
\begin{align*}
\text{Eval (Val n)} &= n \\
\text{Cond e0 e1 e2} &= \text{if \ (Eval e0) \ (Eval e1) \ (Eval e2)}
\end{align*}

For example, the evaluator for singleton booleans states that it actually calculates the boolean denoted by the expression:
\begin{align*}
\text{sEval :: SExpr e \rightarrow SBool (Eval e)} \\
\text{sEval (SVal n) = n} \\
\text{sEval (SCond e0 e1 e2) = sIf (Eval e0) (Eval e1) (Eval e2)}
\end{align*}

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 instructions 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:

\textbf{data Inst (initial :: [Bool]) (final :: [Bool]) where}
\begin{align*}
\text{PUSH :: Sing v \rightarrow Inst s (v : \ s)} \\
\text{IFPOP :: ListInst s st \rightarrow ListInst s sf} \\
&\rightarrow \text{Inst (b : s) (If b st sf)}
\end{align*}

\textbf{data ListInst (initial :: [Bool]) (final :: [Bool]) where}
\begin{align*}
\text{Nil :: ListInst i i} \\
\text{ Nil ++ ys = ys} \\
\text{Nil ++ ys = x :: (xs ++ ys)}
\end{align*}

\textbf{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.

\textbf{compile :: \forall (e :: Expr) (s :: [Bool]).}
\texttt{SExpr e \rightarrow 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.
\begin{align*}
\text{compile } (S\text{Val } y) &= \text{PUSH } y ::: \text{Nil} \\
\end{align*}

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.

\begin{align*}
\text{compile } (S\text{Cond } se0 \ se1 \ se2) &= \\
&= \text{compile } se0 ++ \\
&\quad \text{IFPOP } (\text{compile } se1) (\text{compile } se2) ::: \text{Nil} \\
\end{align*}

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

\begin{align*}
&(\text{If } (\text{Eval } e0) (\text{Eval } e1) (\text{Eval } e2)) :: \text{vs} ::\sim; \text{vs} \\
&(\text{If } (\text{Eval } e0) ((\text{Eval } e1) :: \text{vs}) ((\text{Eval } e2) :: \text{vs})) \\
\end{align*}

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 \text{If}. Therefore, unification cannot be used to instantiate these arguments, so the \text{Proxy} type is necessary.

\begin{align*}
\text{fact} :: \forall t f s b. \text{Sing } b \rightarrow \text{Proxy } t \rightarrow \text{Proxy } f \rightarrow \text{Proxy } s \\
&\rightarrow ((\text{If } b t f) :: s) ::\sim; ((\text{If } b (t :: s) (f :: s)) \\
\text{fact } S\text{True } _ _ _ &= \text{Refl} \\
\text{fact } S\text{False } _ _ _ &= \text{Refl} \\
\end{align*}

We can call \text{fact} in the case for \text{compile}, by providing the appropriate \text{Proxy} arguments.

\begin{align*}
\text{compile } (S\text{Cond } se0 \ (se1 :: \text{Sing } e1) \ (se2 :: \text{Sing } e2)) &= \\
&= \text{case } \text{fact } (s\text{Eval } se0) (\text{Proxy } :: \text{Proxy } (\text{Eval } e1)) \\
&\quad (\text{Proxy } :: \text{Proxy } (\text{Eval } e2)) (\text{Proxy } :: \text{Proxy } s) \text{ of} \\
&\quad \text{Refl } \rightarrow \text{compile } se0 ++ \\
&\quad \text{IFPOP } (\text{compile } se1) (\text{compile } se2) ::: \text{Nil} \\
\end{align*}

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

\begin{align*}
\text{fact } ’ :: \forall t f s b. \text{Sing } b \rightarrow \text{Proxy } t \rightarrow \text{Proxy } f \\
&\rightarrow ((\text{If } b t f) :: s) ::\sim; ((\text{If } b (t :: s) (f :: s)) \\
\text{fact } ’ S\text{True } _ _ _ &= \text{Refl} \\
\text{fact } ’ S\text{False } _ _ _ &= \text{Refl} \\
\end{align*}

However, that version of \text{fact} is even more difficult to use. Because the result of \text{fact } ‘ is used as the argument of \text{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:

\[
\text{compile } (\text{SCond } (\text{se0 :: Sing } e0) \\
(\text{se1 :: Sing } e1) (\text{se2 :: Sing } e2)) = \\
\text{case } \text{fact } '(\text{sEval } se0) \\
(\text{Proxy :: Proxy } (\text{Eval } e1)) \\
(\text{Proxy :: Proxy } (\text{Eval } e2)) :: \\
(\text{if } (\text{Eval } e0) (\text{Eval } e1) (\text{Eval } e2)) ' : s) :~: \\
(\text{if } (\text{Eval } e0) ((\text{Eval } e1) ' : s) ((\text{Eval } e2) ' : s)) \text{ of} \\
\text{Refl } \rightarrow \text{compile } se0 ++ \\
\text{IFPOP } (\text{compile } se1) (\text{compile } se2) :: \text{Nil}
\]

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

\[
\text{fact :: } \forall \ t \ f \ s \ b. \ \text{Sing } b \rightarrow \\
(\text{if } b \ t \ f) ' : s) :~: (\text{if } b (t ' : s) (f ' : s)) \\
\text{fact STrue } = \text{Refl} \\
\text{fact SFalse } = \text{Refl}
\]

and supply the type arguments visibly:

\[
\text{compile } (\text{SCond } se0 (\text{se1 :: Sing } e1) (\text{se2 :: Sing } e2)) = \\
\text{case } \text{fact } @(\text{Eval } e1) @(\text{Eval } e2) @(s \text{Eval } se0) \text{ of} \\
\text{Refl } \rightarrow \text{compile } se0 ++ \\
\text{IFPOP } (\text{compile } se1) (\text{compile } se2) :: \text{Nil}
\]

\section{Integrating visible type application with GHC}

Below, we describe interactions between visible type application and
other features of GHC and some possible extensions enabled by our implementa-

\subsection{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 expression

\[
\text{if } \text{condition then id else } (\lambda x \rightarrow x)
\]

Here, \text{id} has a specified polytype of \( \forall \ a. \ a \rightarrow a \), but the expression \( \lambda x \rightarrow x \) 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 $\lambda x \rightarrow x$ and then choose $\forall a. a \rightarrow a$ as the common supertype of itself and $\forall \{a\}. a \rightarrow a$. However, that may not be possible in general, as there may not always be a common instance of both types.

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:

```haskell
checkIf :: Bool \rightarrow (\forall a. a \rightarrow a) \rightarrow (Bool, Int)
checkIf b = if True then \lambda f \rightarrow (f True, f 5)
else \lambda f \rightarrow (f False, f 3)
```

### B.2 Type classes

Consider the Monad type class:

```haskell
class Applicative m \Rightarrow Monad m where
  return :: a \rightarrow m a
  (>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
...
```

The return and $(\gg=)$ functions have user-supplied type signatures and thus have specified type parameters. But in what order? Our implementation uses the following simple rule: class variables come before method variables. The full types of these methods are thus

```haskell
return :: \forall m. Monad m \Rightarrow \forall a. a \rightarrow m a
(\gg=) :: \forall m. Monad m \Rightarrow \forall a b. m a \rightarrow (a \rightarrow m b) \rightarrow m b
```

Note that, in the type of return, $m$ is quantified before $a$, even though $a$ appears first in the user-supplied type.

### B.3 Instantiate types when inferring

When a variable is defined without an explicit type annotation, are its parameters specified or generalized? According to the systems laid out in this paper, the answer depends on the variable’s definition. For example:

```haskell
id :: a \rightarrow a
id x = x
myId = id
```
According to our technique of lazy instantiation, the use of \textit{id} in the body of \textit{myId} is not instantiated. The variable \textit{myId} thus gets the same type – with its specified type parameter – of \textit{id}.

However, in GHC, we add an extra step: when inferring the type of a variable, deeply instantiate the right-hand side before generalizing. In this example, \textit{id} would get deeply instantiated (that is, all top-level type parameters variables and all type parameters to the right of arrows) before generalizing, giving \textit{myId} a type \(\forall \{a\}. a \rightarrow a\), where the type parameter has been generalized.

This design choice solves all three of the following problems:

– Haskell includes the \textit{monomorphism restriction}. This restriction states that no variable (as distinct from function, which is defined by pattern-matching on arguments) may have a type that includes a class constraint. Consider the definition \textit{myAbs} = \textit{abs}. The type of \textit{abs} is \(\forall a. \text{Num} a \Rightarrow a \rightarrow a\). According to the monomorphism restriction, \textit{myAbs} is not allowed to have this type – it should get the type \textit{Integer} \rightarrow \textit{Integer} according to Haskell’s defaulting rules. Yet, because of lazy instantiation, the type checker never really reasons about the \textit{Num} constraint and would allow \textit{myAbs} to have such the wrong, polymorphic type. By instantiating deeply and then generalizing, the type checker is given a chance to notice the \textit{Num} constraint and react accordingly.

– Haskellers are used to prenex quantification, where all the \(\forall\)s are at the top. But if we say \(x = \lambda_\_ \rightarrow \text{id}\), the naive interpretation of System SB would give \(x :: \forall \{a\}. a \rightarrow \forall b. b \rightarrow b\). This type is unexpected. Documentation for \(x\) would include such a strange type, and clients would have to know to supply the first term-level argument before visibly instantiating the second. By deeply instantiating before generalizing, we give \(x\) the type \(\exists \{a b\}. a \rightarrow b \rightarrow b\), which is more in line with expectations.

– According to our design decision about imported functions in Sect. 3.1 imported functions have specified type parameters if and only if the function is defined with a user-supplied type signature. However, the algorithm in System SB gives \textit{myId}, above, the type \(\forall a. a \rightarrow a\), with a specified type parameter. This would mean that \textit{myId} would be available for visible type application in its defining module, but any importing modules would see a generalized type parameter for \textit{myId}. This discrepancy does not cause any great problems, but it would be unexpected for users. Once again, instantiating and regeneralizing solves the problem.

\subsection{Overloaded numbers}

Numeric literals in Haskell are overloaded. That is, when we write the number 3 in code, it can have any type that is a member of the \textit{Num} class; the type of 3 is thus \(\text{Num} a \Rightarrow a\). It is thus expected that users could write 3 \texttt{@Int} to get the \texttt{Int} 3, instead of an overloaded 3.

However, this does not work. Here is the partial definition of the \textit{Num} class:
class Num a where

    ... fromInteger :: Integer → a

When a 3 is written in code, it gets translated to fromInteger 3, where 3 is our rendering of the Integer 3. According to our treatment of class methods, the full type of fromInteger is ∀ a. Num a ⇒ Integer → a. This type means that any visible type application for an overloaded number literal would have to come between the fromInteger and the number; no straightforward translation from Haskell source could accommodate this. If fromInteger were not a class method, we could just define it to have the type Integer → ∀ a. Num a ⇒ a, which would work nicely. This option, however, is not available.

Happily, it is almost as easy to write (3 :: Int) as 3 @ Int, and so we will go to no great pains to correct this infelicity.

B.5 Ramifications in GHCi

The interactive interpreter GHCi allows users to query the type of expressions. Consider what the answer to the following query should be:

    λ> let myPair :: ∀ a. a → ∀ b. b → (a, b)
    myPair = (,)
    λ> :t myPair 3

It would be reasonable to respond ∀ b. b → (a, b), recalling that numbers are overloaded and we have not yet determined the type of 3. However, this output loses the critical information that the constraint Num a must be satisfied. Alternatively, we could generalize before printing, producing ∀ a b. Num a ⇒ b → (a, b), but that could mislead users into thinking that a is still available for visible type application.

We thus have implemented a middle road, producing ∀ b. Num a ⇒ b → (a, b) in this situation. Note that there is no ∀ a, as a is not available for visible type application. The Num a constraint is listed after the ∀ b quantification only because Haskellers often include type variable quantification before constraints. The precise location of Num a is in fact irrelevant, as there is no facility for visible dictionary application.

B.6 Further 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 GADT

data G a where
    MkG :: ∀ b. G (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:

```plaintext
case g of
  MkG @b → ...
```

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 [31]:

```plaintext
pr :: ∀ (a :: k₁ → k₂) (b :: k₁). Proxy (a b) → Proxy a
pr _ = 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.

### C Properties of System HMV

**Lemma 11 (Inversion for $\leq_{hmv}$).** $\sigma_1 \leq_{hmv} \sigma_2$ if and only if $\sigma_1 = ∀\{\pi_1\}, \bar{b}_1, \bar{b}_2, \pi_1$ and $\sigma_3 = ∀\{\pi_2\}, \bar{b}_2, \tau_2$ where $\tau_1[\pi_1/\bar{b}_1][\pi_1'/\bar{b}_1] = \tau_2$.

**Proof.** By unfolding definitions.

**Lemma 12 (Reflexivity for $\leq_{hmv}$).** Forall $\sigma$, $\sigma \leq_{hmv} \sigma$

**Proof.** By definition.

**Lemma 13 (Transitivity for $\leq_{hmv}$).** If $\sigma_1 \leq_{hmv} \sigma_2$ and $\sigma_2 \leq_{hmv} \sigma_3$, then $\sigma_1 \leq_{hmv} \sigma_3$.

**Proof.** Let $\sigma_3 = ∀\{\pi_3\}, \bar{b}_3, \tau_3$. Then, by inversion, we know $\sigma_2 = ∀\{\pi_2\}, \bar{b}_2, \tau_2$ and $\tau_1[\pi_2/\bar{b}_2][\pi_2'/\bar{b}_2] = \tau_3$. We further know $\sigma_1 = ∀\{\pi_1\}, \bar{b}_1, \bar{b}_1, \tau_1$ and $\tau_1[\pi_1/\bar{b}_1][\pi_1'/\bar{b}_1] = \tau_2$. Thus, $\tau_1[\pi_1/\bar{b}_1][\pi_1'/\bar{b}_1][\pi_2/\bar{b}_2][\pi_2'/\bar{b}_2] = \tau_3$. By the Barendregt convention, we know that $\pi_2$ do not appear in $\tau_1$. Thus we can rewrite as $\tau_1[\pi_2/\pi][\pi_1][\pi_1'/\bar{b}_1][\pi_2/\bar{b}_2][\pi_2'/\bar{b}_2] = \tau_3$. This is enough to finish the derivation via HMV_INSTS and HMV_INSTG.

**Lemma 14 (Substitution in $\leq_{hmv}$).**

1. If $v_1 \leq_{hmv} v_2$, then $v_1[\tau/a] \leq_{hmv} v_2[\tau/a]$.
2. If $\sigma_1 \leq_{hmv} \sigma_2$, then $\sigma_1[\tau/a] \leq_{hmv} \sigma_2[\tau/a]$.
Lemma 15 (Context Generalization for HMV). If $\Gamma \vdash_{\text{hmv}} e : \sigma$ and $\Gamma' \leq_{\text{hmv}} \Gamma$, then $\Gamma' \vdash_{\text{hmv}} e : \sigma$.

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

Proof (Proof of Lemma 16 (Extra knowledge)). This is a corollary of contexts generalization as $\forall \pi. \tau \leq_{\text{hmv}} \forall \pi. \tau$.

D Proofs about System V

Proof (Proof of Soundness of V against HMV (Theorem 7)). 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 $\forall \pi. \tau \leq_{\text{hmv}} \tau[\pi/\bar{\pi}]$ by HMV_INSTS.

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

Lemma 16 (Context generalization for V).

1. If $\Gamma \vdash_{\text{V}} e : v$ and $\Gamma' \leq_{\text{hmv}} \Gamma$, then there exists $v'$ such that $\Gamma' \vdash_{\text{V}} e : v'$ and $v' \leq_{\text{hmv}} v$.
2. If $\Gamma \vdash_{\text{V}} e : \tau$ and $\Gamma' \leq_{\text{hmv}} \Gamma$, then $\Gamma' \vdash_{\text{V}} e : \tau$.
3. If $\Gamma \vdash_{\text{V}} \text{gen} e : \sigma$ and $\Gamma' \leq_{\text{hmv}} \Gamma$, then there exists $\sigma'$ such that $\Gamma' \vdash_{\text{V}} \text{gen} e : \sigma'$ and $\sigma' \leq_{\text{hmv}} \sigma$.

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:

\[
\frac{x : \forall \{\pi\}, v \in \Gamma}{\Gamma \vdash_{\text{V}} x : v[\pi/\bar{\pi}]} - \text{VAR}
\]

Given $x : \forall \{\pi\}, v' \in \Gamma'$ where $\forall \{\bar{\pi}\}, v' \leq_{\text{hmv}} \forall \{\bar{\pi}\}, v$, we must choose $\tau'$ such that $v'[\tau'/\bar{\tau}] \leq_{\text{hmv}} v[\pi/\bar{\pi}]$. Inverting $\leq_{\text{hmv}}$ gives us that $v'[\tau'/\bar{\tau}] \leq_{\text{hmv}} v$. We are thus done by Lemma 14. Note that the size of both derivations is 1.

Case V_TAPP:

\[
\frac{\Gamma \vdash \tau}{\Gamma \vdash e : \forall a. v} - \text{TAPP}
\]

\[
\frac{\Gamma \vdash e : \forall a. v}{\Gamma \vdash e \circ \tau : v[\tau/\bar{\tau}]} - \text{TAPP}
\]
The induction hypothesis gives us $\Gamma \vdash^\varphi e : v'$ where $v' \leq_{hmv} \forall a. v$. By the definition of $\leq_{hmv}$, $v'$ must also be quantified over $a$. We can thus reduce to $\forall a. v'' \leq_{hmv} \forall a. v$, which reduces to $v'' \leq_{hmv} v$. We must prove that $v''[\tau/a] \leq_{hmv} v[\tau/a]$, which follows directly from $v'' \leq_{hmv} v$ via Lemma 14 and so we are done.

**Case $V\_GEN$:**

$$\frac{\overline{\sigma} = ftv(v) \setminus ftv(\Gamma)}{\Gamma \vdash^\sigma e : \forall\{\overline{\sigma}\}. v} V\_GEN$$

The induction hypothesis gives us $\Gamma \vdash^\varphi e : v'$ where $v' \leq_{hmv} v$. By inversion, we know that $v' = \forall\overline{\sigma}', \overline{\tau}_1$ and $v = \forall\overline{\sigma}'. \tau_2$ where $\tau_1[\overline{\tau}/\overline{\sigma}] = \tau_2$.

Let $\overline{\sigma} = ftv(v) \setminus ftv(\Gamma)$ and $\overline{\tau} = ftv(v') \setminus ftv(\Gamma')$. We want to show that $\forall\{\overline{\tau}\}, v' \leq_{hmv} \forall\{\overline{\sigma}\}. v$. Expanding out, we want to show that $\forall\{\overline{\tau}\}, \overline{\tau}_1 \leq_{hmv} \forall\{\overline{\sigma}\}, \overline{\tau}_2$, where $\tau_1[\overline{\tau}/\overline{\sigma}] = \tau_2$. Unfolding $\leq_{hmv}$ shows that we want $\tau_1[\overline{\tau}/\overline{\sigma}] \leq_{hmv} \tau_2$. Choose $\tau = \overline{\tau}$ and we are done.

**Lemma 17 (Type substitution).** If $\Gamma \vdash \tau$ then $\Gamma[\tau'/a] \vdash \tau$.

**Proof.** By the definition of rule Ty\_SCOPED. Note that substituting in the context has no effect.

**Lemma 18 (Substitution for $V$).** Assume $a \notin \Gamma$. That is, $a$ is not a scoped type variable. Further, assume $\Gamma \vdash \tau$.

1. If $\Gamma \vdash^\varphi e : \tau'$, then $\Gamma[\tau/a] \vdash^\varphi e : \tau'[\tau/a]$.
2. If $\Gamma \vdash^\varphi e : v$, then $\Gamma[\tau/a] \vdash^\varphi e : v[\tau/a]$.
3. If $\Gamma \vdash^\varphi e : \sigma$, then $\Gamma[\tau/a] \vdash^\varphi e : \sigma[\tau/a]$.

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

Note that $a$ cannot appear anywhere in an expression, as $a$ is not a scoped type variable. Thus, any types appearing in expressions are unaffected by the substitution $[\tau/a]$. This realization covers the $V\_TAPP$ case.

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

The induction hypothesis gives us $\Gamma[\tau/a] \vdash^\varphi e : v[\tau/a]$. Let $\overline{\sigma} = ftv(v[\tau/a]) \setminus ftv(\Gamma[\tau/a])$. To use $V\_GEN$ to conclude $\Gamma \vdash^\varphi e : (\forall\{\overline{\sigma}\}. v)[\tau/a]$, we must show that $\overline{\tau} = \overline{\sigma}$ and that $\overline{\sigma}$ are not free in $\tau$.

We now have several cases:

- **a is free in both $v$ and in $\Gamma$:** In this case $ftv(v[\tau/a])$ includes the free type variables of $v$, minus $a$, plus the free variables of $\tau$. Likewise, $ftv(\Gamma[\tau/a])$ includes $ftv(\Gamma)$, minus $a$, plus the free variables of $\tau$. In each case, the sets that produce $\overline{\sigma}$ are changed by the same variables. Therefore, $\overline{\tau}$ and $\overline{\sigma}$ must be equal.
Proof (Proof of Completeness (Theorem $[\overline{\text{6}}]$)). We proceed by induction on $\Gamma \vdash_{\text{hmv}} e : \sigma$.

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

Case HMV_ABS:

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

The induction hypothesis gives us $\Gamma, x : \tau_1 \vdash_{\text{hmv}} e : \tau_2$ where $\tau_2 = \tau'_2[\overline{\pi} / \overline{\overline{\pi}}]$. Inverting $\vdash_{\text{hmv}}^{\text{gen}}$ gives us $\Gamma, x : \tau_1 \vdash^{\text{gen}} e : \forall \overline{\pi} \tau_2$. We can then use $V_{\text{INSTS}}$ and $V_{\text{ABS}}$ to get $\Gamma \vdash_{\text{hmv}} \lambda x. e : \tau_1 \to \tau'_2[\overline{\pi} / \overline{\overline{\pi}}]$. Generalizing, we get $\Gamma \vdash^{\text{gen}}_{\text{hmv}} \lambda x. e : \forall \overline{\pi} \tau_1 \to \tau'_2[\overline{\pi} / \overline{\overline{\pi}}]$ where the new variables $\overline{\pi'}$ come from generalizing $\tau_1$ and the $\overline{\pi'}$. We can see that $(\tau_1 \to \tau'_2[\overline{\pi} / \overline{\overline{\pi}}])[\overline{\overline{\pi}} / \overline{\pi}] = \tau_1 \to \tau_2$ and so $\forall \overline{\pi} \overline{\pi'} \tau_1 \to \tau'_2[\overline{\pi} / \overline{\overline{\pi}}] \leq_{\text{hmv}} \tau_1 \to \tau_2$ and we are done.

Case HMV_APP:

$$\frac{\Gamma \vdash_{\text{hmv}} e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\text{hmv}} e_2 : \tau_1 \quad \text{HMV_APP}}{\Gamma \vdash_{\text{hmv}} e_1 e_2 : \tau_2}$$

The induction hypothesis gives us $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_1 : \forall \overline{\pi_1} \tau_1 \to \tau_{12}$ with $\tau_1 = \tau_{11}[\overline{\pi_1}/\overline{\overline{\pi_1}}][\overline{\tau_1}/\overline{\overline{\tau_1}}]$ and $\tau_2 = \tau_{12}[\overline{\tau_1}/\overline{\overline{\tau_1}}][\overline{\tau'_1}/\overline{\overline{\tau'_1}}]$, along with $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_2 : \forall \overline{\pi_2} \tau_{12} \to \tau_{21}$ with $\tau_1 = \tau_{21}[\overline{\tau_2}/\overline{\overline{\tau_2}}][\overline{\tau'_2}/\overline{\overline{\tau'_2}}]$. Inverting $\vdash_{\text{hmv}}^{\text{gen}}$ gives us $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_1 : \forall \overline{\pi_1} \tau_{11} \to \tau_{12}$ and $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_2 : \forall \overline{\pi_2} \tau_{21}$. We now use the Substitution Lemma (Lemma $[\text{18}]$) with the substitution $[\overline{\pi_1}/\overline{\pi_1}]$ on the first of these to yield $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_1 : \forall \overline{\pi_1} \tau_{11}[\overline{\pi_1}/\overline{\overline{\pi_1}}] \to \tau_{12}[\overline{\tau_1}/\overline{\overline{\tau_1}}]$. Note that the $\overline{\pi_1}$ must not be free in $\Gamma$, by inversion of $\vdash_{\text{hmv}}^{\text{gen}}$. Similarly, Lemma $[\text{18}]$ gives us $\Gamma \vdash_{\text{hmv}}^{\text{gen}} e_2 : \forall \overline{\pi_2} \tau_{21}[\overline{\tau_2}/\overline{\overline{\tau_2}}]$.

We can then use $V_{\text{INSTS}}$ on both of these judgments, to show $\Gamma \vdash e_1 : \tau_{11}[\overline{\pi_1}/\overline{\overline{\pi_1}}][\overline{\tau_1}/\overline{\overline{\tau_1}}] \to \tau_{12}[\overline{\tau_1}/\overline{\overline{\tau_1}}][\overline{\tau'_1}/\overline{\overline{\tau'_1}}]$ and $\Gamma \vdash e_2 : \tau_{12}[\overline{\tau_2}/\overline{\overline{\tau_2}}][\overline{\tau'_2}/\overline{\overline{\tau'_2}}]$.

We can now use $V_{\text{APP}}$, as the argument type is equal to $\tau_1$, established earlier. Rule $V_{\text{APP}}$ then gives us $\Gamma \vdash e_1 e_2 : \tau_{12}[\tau_{11}/\overline{\overline{\tau_1}}][\overline{\tau'_1}/\overline{\overline{\tau'_1}}]$. This type, as noted earlier, equals $\tau_2$, and so we are done.

Case HMV_INT

$$\frac{\Gamma \vdash_{\text{hmv}} n : \text{Int}}{\text{HMV_INT}}$$

Trivial.
Case HMV\_TApp:

\[
\Gamma \vdash \tau \\
\Gamma \vdash_{\text{hmv}} e : \forall a.\upsilon \\
\frac{}{\Gamma \vdash_{\text{hmv}} e \mid \tau : \upsilon[\tau/a]} \text{HMV\_TApp}
\]

The induction hypothesis (after inverting $\forall e$) gives us $\Gamma \vdash e : \forall a.\upsilon'$, where $\bar{b} = \text{ftv}(\forall a.\upsilon') \setminus \text{ftv}(\Gamma)$ and $\upsilon'[\bar{b}] \leq_{\text{hmv}} \upsilon$. Applying V\_TApp gives us $\Gamma \vdash e \mid \tau : \upsilon'[\tau/a]$, and V\_Gen gives us $\Gamma \vdash e : \forall \upsilon[\tau/a] \leq_{\text{hmv}} \upsilon[\tau/a]$, which follows when there is some $\bar{\tau}$, such that $\upsilon'[\tau/a][\bar{\tau}] \leq_{\text{hmv}} \upsilon[\tau/a]$. This is equivalent to exchanging the substitution, i.e. finding a $\bar{\upsilon}'$ such that $\upsilon'[\bar{\tau}/\bar{\upsilon}] \leq_{\text{hmv}} \upsilon[\tau/a]$.

By Substitution (Lemma [14]), we have $\upsilon'[\bar{\tau}/\bar{\upsilon}] \leq_{\text{hmv}} \upsilon[\tau/a]$. We also know that the $\bar{b}$ are a subset of the $\upsilon$. So we can choose $\bar{\upsilon}'$ to be $\upsilon$ for the $\bar{b}$, and the remaining $\bar{\upsilon}$ elsewhere, and we are done.

Case HMV\_Let:

\[
\frac{}{\Gamma \vdash_{\text{hmv}} e_1 : \sigma_1 \\
\Gamma, x : \sigma_1 \vdash_{\text{hmv}} e_2 : \sigma_2 \\
\frac{}{\Gamma \vdash_{\text{hmv}} \text{let } x = e_1 \text{ in } e_2 : \sigma_2} \text{HMV\_Let}
\]

The induction hypothesis gives us $\Gamma \vdash e : \forall a.\upsilon$ with $\sigma_1 \leq_{\text{hmv}} \sigma_1$. The induction hypothesis also gives us $\Gamma, x : \sigma_1 \vdash e_2 : \sigma_2$ with $\sigma_2 \leq_{\text{hmv}} \sigma_2$. Use Lemma [10] to get $\Gamma, x : \sigma_1 \vdash \sigma_2$ where $\forall a.\upsilon \leq_{\text{hmv}} \sigma_2$.

Let $\sigma_2 = \forall \{\bar{b}\}.\upsilon$ where $\bar{b} = \text{ftv}(\upsilon) \setminus \text{ftv}(\Gamma)$. Inverting $\forall e$ gives us $\Gamma, x : \sigma_1 \vdash e_2 : \upsilon$. We then use V\_Let to get $\Gamma \vdash_{\text{hmv}} \text{let } x = e_1 \text{ in } e_2 : \upsilon$. Generalizing gives us $\Gamma \vdash_{\text{hmv}} \text{let } x = e_1 \text{ in } e_2 : \forall \{\bar{b}\}.\upsilon$.

Transitivity of $\leq_{\text{hmv}}$ (Lemma [13]) gives us $\forall \{\bar{b}\}.\upsilon \leq_{\text{hmv}} \sigma_2$.

Case HMV\_Annot:

\[
\frac{}{\Gamma \vdash \upsilon \quad \upsilon = \forall \pi, \bar{b}, \tau \\
\Gamma, \pi \vdash_{\text{hmv}} e : \tau \\
\frac{}{\pi, \bar{b} \not\in \text{ftv}(\Gamma)} \text{HMV\_Annot}
\]

The induction hypothesis gives us $\Gamma \vdash \forall e : \forall \bar{b}.\tau'$ with $\upsilon'[\bar{b}][\bar{\tau}][\bar{\pi}] = \tau$. Inverting $\forall e$ gives us $\Gamma \vdash e : \forall \bar{b}.\tau'$.

Applying V\_InstS gives us $\Gamma \vdash e : \tau$ and we can use V\_Annot to be done.

Case HMV\_Int: Trivial.

Case HMV\_Gen:

\[
\frac{}{\Gamma \vdash_{\text{hmv}} e : \sigma \\
\frac{}{a \not\in \text{ftv}(\Gamma)} \text{HMV\_Gen}
\]

The induction hypothesis gives us $\Gamma \vdash e : \forall a.\sigma'$ where $\sigma' \leq_{\text{hmv}} \forall a.\sigma$. We know $\sigma' \leq_{\text{hmv}} \forall a.\sigma$. In other words, if $\sigma' = \forall \{\bar{b}\}, \upsilon_1$ and $\sigma = \forall \{\tau\}, \upsilon_2$, we have some $\bar{\tau}$ such that $\upsilon_1[\tau/b] = \upsilon_2$. By the definition of $\leq_{\text{hmv}}$ we can use these same $\bar{\tau}$ to show that $\sigma' \leq_{\text{hmv}} \forall a.\bar{\tau}$.\upsilon_2.
Case $\text{HMV}_{\text{SUB}}$:

$$
\frac{
\Gamma \vdash_{\text{hmv}} e : \sigma_1 \quad \sigma_1 \leq_{\text{hmv}} \sigma_2
}{
\Gamma \vdash_{\text{hmv}} e : \sigma_2}
$$

$\text{HMV}_{\text{SUB}}$

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

## E Algorithm $\mathcal{V}$

In this appendix, we use metavariables $Q$, $R$, and $S$ to refer to substitutions from type variables $a$ to monotypes $\tau$. 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 $S = \mathcal{U}_\pi(\tau_1, \tau_2)$ then either $S(\tau_1) = S(\tau_2)$ and $\pi \cap \text{dom}(S) = \emptyset$, or no such $S$ exists.
- If $R(\tau_1) = R(\tau_2)$ (and $\pi \cap \text{dom}(R) = \emptyset$) then there exists some $S$ such that $R = S \circ \mathcal{U}_\pi(\tau_1, \tau_2)$. In other words, unification produces the most general unifier.

We also must define a new operation $\text{vars}(\Gamma)$ which extracts the scoped type variables listed in $\Gamma$.

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 19 (Algorithm $V$).

1. $V(\Gamma, \lambda x. e) = (S_1, S_1(b) \rightarrow \tau)$ when $\forall \pi. \pi \in \Gamma$ $S_1 = U_{\text{vars}(\tau)}(S_2(\tau_1), \tau_2 \rightarrow b)$ $b$ fresh

2. $V(\Gamma, e_1 e_2) = (S_3 \circ S_2 \circ S_1, S_3(b))$ when $\forall \pi. \pi \in S_1(\Gamma), e_1$ $S_3 = U_{\text{vars}(\tau)}(S_2(\tau_1), \tau_2 \rightarrow b)$ $b$ fresh

3. $V(\Gamma, n) = (\epsilon, \text{Int})$

4. $V(\Gamma, e) = (S_1, \tau)$ when $\forall \pi. \pi \in S_1(\Gamma), x: \sigma_1$, $\forall \pi. \pi \in S_1(\Gamma)$ $S_2 = U_{\text{vars}(\tau)}(S_2(\tau_1), \tau_2 \rightarrow b)$ $b$ fresh

5. $V^*(\Gamma, x) = (\epsilon, v)$ when $\forall \pi. \pi \in \Gamma$ $\forall \pi. \pi \in \Gamma$

6. $V^*(\Gamma, \text{let } x = e_1 \text{ in } e_2) = (S_2 \circ S_1, v_2)$ when $\forall \pi. \pi \in S_1(\Gamma), x: \sigma_1, e_2$ $S_2 = U_{\text{vars}(\tau)}(S_2(\tau_1), \tau_2 \rightarrow b)$ $b$ fresh

7. $V^*(\Gamma, e \oplus \tau) = (S_1, v_1[\tau/a])$ when $\forall \pi. \pi \in S_1(\Gamma), x: \sigma_1, e_2$ $S_2 = U_{\text{vars}(\tau)}(S_2(\tau_1), \tau_2 \rightarrow b)$ $b$ fresh

8. $V^*(\Gamma, (\Lambda \pi. e : v)) = (S_2 \circ S_1, v)$ when $\forall \pi. \pi \in \Gamma$ $\forall \pi. \pi \in \Gamma$ $\forall \pi. \pi \in \Gamma$ $\forall \pi. \pi \in \Gamma$

9. $V^*(\Gamma, e) = V(\Gamma, e)$

10. $V^{\text{gen}}(\Gamma, e) = (S, \forall \pi. \pi \in \Gamma, v)$ when $\forall \pi. \pi \in \Gamma, S, v \in \Gamma$ $\forall \pi. \pi \in \Gamma$ $\forall \pi. \pi \in \Gamma$

Lemma 20 (Soundness of Algorithm $V$).
1. If $V(\Gamma, e) = (S, \tau)$ then $S(\Gamma) \vdash e : \tau$
2. If $V^*(\Gamma, e) = (S, v)$ then $S(\Gamma) \vdash e : v$
3. If $V_{\text{gen}}(\Gamma, e) = (S, \sigma)$ then $S(\Gamma) \vdash e : \sigma$

In all cases, $\text{dom}(S) \cap \text{vars}(\Gamma) = \emptyset$.

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 $V(\Gamma, \lambda x. e) = (S_1, S_1(b) \to \tau)$, where $(S_1, \tau) = V((\Gamma, x:b), e)$ and $b$ is fresh. We must show $S_1(\Gamma) \vdash \lambda x. e : S_1(b) \to \tau$. The induction hypothesis tells us $S_1(\Gamma, x:b) \vdash e : \tau$. Rewrite this as $S_1(\Gamma, x:S_1(b)) \vdash e : \tau$. Use $\text{V}_\text{Abs}$ then gives us the desired result.

Case $e = e_1 e_2$: By (2), we have $V(\Gamma, e_1 e_2) = (S_3 \circ S_2 \circ S_1, S_3(b))$, with several side conditions from the statement of $V$. Let $R = S_2 \circ S_1$. We must show $R(\Gamma) \vdash e_1 e_2 : S_3(b)$. The induction hypothesis gives us $S_1(\Gamma) \vdash e_1 : \tau_1$ and $S_2(\Gamma) \vdash e_2 : \tau_2$. Furthermore, we know that $S_3(S_2(\tau_1)) = S_3(\tau_2 \to b)$.

By the substitution lemma (Lemma 18), we know that $R(\Gamma) \vdash e_1 : S_3(S_2(\tau_1))$ and $R(\Gamma) \vdash e_2 : S_3(\tau_2)$. The first of these can be rewritten to $R(\Gamma) \vdash e_1 : S_3(\tau_2 \to b)$, or $R(\Gamma) \vdash e_1 : S_3(\tau_2) \to S_3(b)$. We now use $\text{V}_\text{App}$ to get $R(\Gamma) \vdash e_1 e_2 : S_3(b)$ as desired. Note that $\text{dom}(R) \cap \text{vars}(\Gamma) = \emptyset$ by virtue of the fact that $S_1$ and $S_2$ meet this condition (by the induction hypothesis) and $S_3$ does by the properties of $U$.

Case $e = n$: By (3), we have $V(\Gamma, n) = (\epsilon, \text{int})$. We must prove $\Gamma \vdash n : \text{int}$, which we get from $\text{V}_\text{int}$.

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

2. Case $e = x$: By (5), we know $V^*(\Gamma, x) = (\epsilon, v)$ where $x:V(x[\pi], v \in \Gamma)$. We must show $\Gamma \vdash x : v$. This is direct from $\text{V}_\text{var}$, choosing $\pi = \pi$.

Case $e = \text{let } x = e_1 \text{ in } e_2$: By (6), we know $V^*(\Gamma, \text{let } x = e_1 \text{ in } e_2) = (S_2 \circ S_1, v_2)$ where $(S_1, \sigma_1) = V^*(\Gamma, e_1)$ and $(S_2, v_2) = V^*((S_1, \Gamma), \lambda x : \sigma_1)$. We must show $S_2(S_1(\Gamma)) \vdash e_1 \text{ in } e_2 : v_2$. The induction hypothesis gives us $S_1(\Gamma) \vdash e_1 : \sigma_1$ and $S_2(S_1(\Gamma), \lambda x : \sigma_1) \vdash e_2 : v_2$. Substitution on the former gives us $S_2(S_1(\Gamma)) \vdash e_1 : \sigma_1$ and we can rewrite the latter as $S_2(S_1(\Gamma), \lambda x : \sigma_1) \vdash e_2 : v_2$. Use $\text{V}_\text{let}$ gives us our desired result.

Case $e = e_0 \circ e_1$: By (7), we know $V^*(\Gamma, e_0 \circ e_1) = (S_1, v_1[\tau/b])$ where $(S_1, \forall v. v_1) = V^*(\Gamma, e)$. We must show $S_1(\Gamma) \vdash e_0 : \forall v. v_1 \land f$. The induction hypothesis gives us $S_1(\Gamma) \vdash e_0 : v_1$, and we are done by $\text{V}_\text{app}$, noting that $\Gamma \vdash \tau$ implies $S_1(\Gamma) \vdash \tau$.

Case $e = (\varpi, e_0 : v)$: By (8), we know $V^*(\Gamma, (\varpi, e_0 : v)) = (S_2 \circ S_1, v)$ with side conditions from the statement of $V$, including $\forall \varpi, \overline{\varpi}, \tau = v$. We must show $S_2(S_1(\Gamma)) \vdash (\varpi, e_0 : v) : v$. The induction hypothesis
Lemma 21 (Substitution/generalization). \[ \forall \{ S \} \text{simplifies to } \subseteq \text{ftv} S \] Proof. We must show \( Q \). by reflexivity of \( \text{dom} \) that are conjured up as fresh during recursive calls. However, \( \text{vars}(\Gamma), \pi, \emptyset \cap \text{dom}(S_2) = \emptyset \), so \( S_2(\tau) = \tau \). We thus can use \( \text{V}_{\text{ANNOT}} \) and we are done.

Other cases: By (9), we have \( V^*(\Gamma, e) = (S, \tau) \) where \( (S, \tau) = V(\Gamma, e) \).

We must show \( S(\Gamma) \vdash e : \tau \). The induction hypothesis gives us \( S(\Gamma) \vdash e : \tau \). We are done by \( \text{V}_{\text{MONO}} \).

3. All cases: By (10), we know \( V^\text{gen}(\Gamma, e) = (S, \forall \{ \pi \}, \nu) \) where \( (S, \nu) = V^*(\Gamma, e) \) and \( \pi = \text{ftv}(\nu) \setminus \text{ftv}(S(\Gamma)) \). We must show \( S(\Gamma) \vdash^\text{gen} e : \forall \{ \pi \}. \nu \).

The induction hypothesis gives us \( S(\Gamma) \vdash e : \nu \), and we are done by \( \text{V}_{\text{GEN}} \).

Lemma 21 (Substitution/generalization). If \( \pi = \text{ftv}(\nu) \setminus \text{ftv}(\Gamma) \) and \( \emptyset = \text{ftv}(S(\Gamma)) \), then \( S(\forall \{ \pi \}, \nu) \leq_{\text{hmv}} \forall \{ \emptyset \}, S(\nu) \)

Proof. We must show \( S(\forall \{ \pi \}, \nu) \leq_{\text{hmv}} \forall \{ \emptyset \}, S(\nu) \). Simplify this to \( \forall \{ \pi \}, S(\nu[\pi/\pi]) \leq_{\text{hmv}} \forall \{ \emptyset \}, S(\nu) \) where the \( \pi \) are fresh. (They are used to implement capture-avoidance.) By the definition of \( \leq_{\text{hmv}} \) (HMV\_INSTG), this simplifies to \( S(\nu[\pi/\pi]) \leq_{\text{hmv}} S(\nu) \), for our choice of \( \pi \). Choose \( \pi = S(\pi) \), yielding our wanted to be \( S(\nu[\pi/\pi]) \leq_{\text{hmv}} S(\nu) \). Simplifying again yields \( S(\nu[\pi/\pi]) \leq_{\text{hmv}} S(\nu) \), which is the same as \( S(\nu) \leq_{\text{hmv}} S(\nu) \). We are done by reflexivity of \( \leq_{\text{hmv}} \).

Lemma 22 (Completeness of Algorithm \( V \)). For all contexts \( \Gamma \) and substitutions \( Q \) such that \( \text{dom}(Q) \cap \text{vars}(\Gamma) = \emptyset \):

1. If \( Q(\Gamma) \vdash e : \tau \), then \( V(\Gamma, e) = (S, \tau') \) and there exists \( R \) such that \( Q = R \circ S \) and \( \tau = R(\tau') \).
2. If \( Q(\Gamma) \vdash e : \nu \), then \( V^*(\Gamma, e) = (S, \nu') \) and there exists \( R \) such that \( Q = R \circ S \) and \( \nu = R(\nu') \).
3. If \( Q(\Gamma) \vdash^\text{gen} e : \sigma \), then \( V^\text{gen}(\Gamma, e) = (S, \sigma') \) and there exists \( R \) such that \( Q = R \circ S \) and \( R(\sigma') \leq_{\text{hmv}} \sigma \).

In the \( Q = R \circ S \) conclusions above, we ignore any differences on type variables that are conjured up as fresh during recursive calls. However, we require that \( \text{dom}(R) \cap \text{vars}(\Gamma) = \emptyset \).

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 \( V \) (such as \( (S, \tau') \))
(ii) The substitution \( R \)
(iii) The fact that \( \text{dom}(R) \cap \text{vars}(\Gamma) = \emptyset \)
(iv) The fact that \( Q = R \circ S \)
(v) The fact that, say, \( \tau = R(\tau') \)

These pieces will be labeled in each case.
Case $V_{\text{ABS}}$:

$$
\Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash \epsilon_1 : \tau_2 \\
\Gamma \vdash \epsilon_2 : \tau_1 \\
\Gamma \vdash \epsilon_1 \epsilon_2 : \tau_2 \\
$$

We know that $Q(\Gamma), x : \tau_1 \vdash e : \tau_2$. Let $\Gamma' = \Gamma, x : b$ (where $b$ is fresh) and $Q' = [\tau_1/b] \circ Q$. Then, we can see that $Q'(\Gamma') \vdash e : \tau_2$. We thus use the induction hypothesis to get $V(\Gamma, x : b, e) = (S, \tau'_2)$ along with $R'$ such that $Q' = R' \circ S$ and $\tau_2 = R'(\tau'_2)$. We can thus see that

(i) $V(\Gamma, \lambda x. e) = (S, S(b) \rightarrow \tau'_2)$.

We have left only to provide $R$ such that $Q = R \circ S$ and $\tau_1 \rightarrow \tau_2 = R(S(b) \rightarrow \tau'_2) = R(S(b)) \rightarrow R(\tau'_2)$.

(ii) Choose $R = R'$.

(iii) The domain restriction follows by the induction hypothesis.

By functional extensionality, $Q = R' \circ S$ if $Q$ 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

(iv) $Q = Q' = R' \circ S$, ignoring the fresh $b$.

(v) 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 $V_{\text{APP}}$:

$$
\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \\
\Gamma \vdash \epsilon_1 : \tau_2 \\
\Gamma \vdash \epsilon_2 : \tau_1 \\
\Gamma \vdash \epsilon_1 \epsilon_2 : \tau_2 \\
$$

The induction hypothesis tells us that $V(\Gamma, e_1) = (S_1, \tau'_1)$ with $R_1$ such that $Q = R_1 \circ S_1$ and $\tau_1 \rightarrow \tau_2 = R_1(\tau'_1)$. Recall that we are assuming $Q(\Gamma) \vdash e_1 : \tau_2$, which can now be written as $R_1(S_1(\Gamma)) \vdash e_1 : \tau_2$. We thus know (by inversion) $R_1(S_1(\Gamma)) \vdash e_2 : \tau_1$. 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 $V(S_1(Q), e_2) = (S_2, \tau'_2)$ 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 $\tau'_2 \rightarrow b$ for some fresh $b$. We know $R_1(\tau'_1) = \tau_1 \rightarrow \tau_2$ and $R_2(\tau'_2) = \tau_1$. We can rewrite the former as $R_2(S_2(\tau'_1)) = \tau_1 \rightarrow \tau_2$.

Choose $S'_3 = [\tau_2/b] \circ R_2$. We see that $S'_3(S_2(\tau'_1)) = S'_3(\tau'_2 \rightarrow b)$ as required. We also must show that $\text{dom}(S'_3) \cap \text{vars}(\Gamma) = \emptyset$. This comes from the fact that $b$ is fresh and that $R_2$ satisfies the domain restriction by the induction hypothesis. We now know that $U_{\text{vars}(\Gamma)}(S_2(\tau'_1), \tau'_2 \rightarrow b)$ will succeed with the most general unifier $S_3$. We thus know that

(i) $V(\Gamma, e_1, e_2) = (S_1 \circ S_2 \circ S_1, S_3(b))$.

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 by the most-general-unifier property.

(iii) $R$ must satisfy the domain restriction from the fact that both $S_3$ and $S'_3$ do.
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 $\tau_2 = R(S_3(b)) = S_3'(b)$.

This comes from the definition of $S_3$.

We are done.

**Case V **_**INT:**

\[ \Gamma \vdash_\psi \text{n : Int} \]

(i) $\forall (\Gamma, \text{n}) = (\epsilon, \text{Int})$.

(ii) Choose $R = Q$.

(iii) By the assumed domain restriction on $Q$.

(iv) $Q = R \circ \epsilon$, quite easily.

(v) $Q(\text{Int})$ sure does equal $\text{Int}$.

**Case V **_**INSTS:**

\[ \Gamma \vdash_\psi \text{e : } \forall \bar{\alpha}. \tau \]

no other rule matches

\[ \Gamma \vdash_\psi \text{e : } \tau[\bar{\tau}/\bar{\alpha}] \]

The induction hypothesis gives us $V^*(\Gamma, \epsilon) = (S, \forall \bar{\alpha}. \tau')$ with $R'$ such that $Q = R' \circ S$ and $R'('\forall \bar{\alpha}. \tau') = \forall \bar{\alpha}. \tau$. Note that we have liberally renamed bound variables here to ensure that the quantified variables $\bar{\alpha}$ are the same in both cases. This is surely possible because the substitution $R'$ cannot change the number of quantified variables (noting that $\tau'$ must not have any quantified variables itself). We thus know $R'(\tau') = \tau$ and that $\bar{\alpha} \cap \text{dom}(R') = \emptyset$.

(i) We can see that $\forall (\epsilon, \text{Int})$.

(ii) Choose $R$ to be the $[\tau/\bar{\alpha}] \circ R'$.

(iii) The domain restriction is satisfied by the induction hypothesis and by the Barendregt convention applied to bound variables $\bar{\alpha}$.

(iv) We can see that $Q = R \circ S$ as required (ignoring the action on the fresh $\bar{\alpha}$).

We have already established that $R'(\tau') = \tau$. We must show that $R(\tau') = \tau[\bar{\tau}/\bar{\alpha}]$.

(v) This follows from our definition of $R$.

We are done.

**Case V **_**VAR:**

\[ x: \forall \{\bar{\alpha}\}, \text{v } \in \Gamma \]

\[ \Gamma \vdash_\psi \text{x : v[\bar{\tau}/\bar{\alpha}]} \]

We know $x: \forall \{\bar{\alpha}\}, \text{v } \in Q(\Gamma)$. Thus, there exists $v'$ such that $x: \forall \{\bar{\alpha}\}, v' \in \Gamma$ where $Q(v') = v$.

(i) Thus, $\forall^*(\Gamma, x) = (\epsilon, v')$.

(ii) Choose $R = [\tau/\bar{\alpha}] \circ Q$.

(iii) The domain restriction is satisfied by the assumption of domain restriction on $Q$ and the Barendregt convention applied to the bound $\bar{\alpha}$.

(iv) Clearly, $Q = R \circ \epsilon$, ignoring the action on the fresh $\bar{\alpha}$.

We must now show that $R(v') = v[\bar{\tau}/\bar{\alpha}]$.

(v) This is true by construction of $R$. 
Case V_Let:

\[
\begin{array}{c}
\Gamma \vdash \text{gen} e_1 : \sigma_1 \\
\Gamma, x : \sigma_1 \vdash e_2 : v_2 \\
\end{array}
\]

\[
\Gamma \vdash e \text{ let } x = e_1 \text{ in } e_2 : v_2
\]

The induction hypothesis gives us \( V^{\text{gen}}(\Gamma, e_1) = (S_1, \sigma'_1) \) with \( R_1 \) such that \( Q = R_1 \circ S_1 \) and \( R_1(\sigma'_1) \leq_{\text{hmv}} \sigma_1 \). We know (from inversion) that \( R_1(S_1(\Gamma)), x : \sigma_1 \vdash e_2 : v_2 \). We also see that \( R_1(S_1(\Gamma), x : \sigma_1) \leq_{\text{hmv}} R_1(S_1(\Gamma)) \), \( x : \sigma_1 \) and thus that \( R_1(S_1(\Gamma), x : \sigma_1) \vdash e_2 : v_2 \), by context generalization (Lemma 16), which preserves heights of derivations. We thus use the induction hypothesis again to get \( V^*((S_1(\Gamma), x : \sigma_1'), e_2) = (S_2, v'_2) \) with \( R_2 \) such that \( R_1 = R_2 \circ S_2 \) and \( v_2 = R_2(v'_2) \).

(i) We thus have \( V^*(\Gamma, \text{let } x = e_1 \text{ in } e_2) = (S_2 \circ S_1, v'_2) \).

(ii) Choose \( R = R_2 \).

(iii) The domain restriction is satisfied via the induction hypothesis.

(iv) We can see that \( Q = R_2 \circ S_2 \circ S_1 \) as desired.

(v) We can further see that \( R_2(v'_2) = v_2 \) as desired.

Case V_TApp:

\[
\begin{array}{c}
\Gamma \vdash \tau \\
\Gamma \vdash e : \forall a. \nu \\
\end{array}
\]

\[
\Gamma \vdash e \circ \tau : v[\tau/a]
\]

The induction hypothesis gives us \( V^*(\Gamma, e) = (S, \nu') \) with \( R' \) such that \( Q = R' \circ S \) and \( \forall a. \nu = R'(\nu') \). Because the substitution \( R' \) maps type variables only to monotypes, we know \( \nu' \) must be \( \forall a. \nu'' \), with \( R'(\nu'') = \nu \) and \( a \notin \text{dom}(R') \).

(i) We thus know \( V^*(\Gamma, e \circ \tau) = (S, \nu''[\tau/a]) \).

(ii) Choose \( R = R' \).

(iii) The domain restriction is satisfied via the induction hypothesis.

(iv) We already know \( Q = R' \circ S \).

We must show \( R'(\nu''[\tau/a]) = \nu[\tau/a] \). This can be reduced to \( R'(\nu'')[\tau/a] = \nu[\tau/a] \) by the fact that \( a \notin \text{dom}(R) \). Furthermore, we know \( \tau \) is closed, so we can further reduce to \( R'(\nu'')[\tau/a] = \nu[\tau/a] \).

(v) But we know \( R'(\nu'') = \nu \), so we are done.

Case V_Annot:

\[
\begin{array}{c}
\Gamma \vdash v \\
\Gamma, \overline{\pi} \vdash e : \tau \\
\end{array}
\]

\[
\Gamma \vdash (\Lambda \overline{\pi}. e : v) : v
\]

The induction hypothesis gives us \( V((\Gamma, \overline{\pi}), e) = (S_1, \tau') \) with \( R_1 \) such that \( Q = R_1 \circ S_1 \), \( R_1(\tau') = \tau, \) and \( \text{dom}(R_1) \cap \text{vars}(\Gamma, \overline{\pi}) = \emptyset \). We must show that \( \tau \) and \( \tau' \) have a unifier. We can assume further (by the Barendregt convention) that \( \overline{b} \cap \text{dom}(R_1) = \emptyset \). We also know that \( \text{ftv}(\tau) \subseteq \text{vars}(\Gamma), \overline{\pi}, \overline{b} \). Thus, \( R_1(\tau) = \tau \) and \( R_1 \) is a unifier of \( \tau \) and \( \tau' \). Let \( S_2 = \mathcal{U}_{\text{vars}(\Gamma), \overline{\pi}}(\tau, \tau') \) (which is now sure to exist).
(i) We thus have $V^*(\Gamma, (\Lambda \pi. e: \nu)) = (S_2 \circ S_1, \nu)$.
(ii) Choose $R$ as determined by the fact that $R_1 = R \circ S_2$ (gained from the most-general-unifier property).
(iii) The domain restriction comes from the fact that $S_2$ and $R_1$ both satisfy even a stricter domain restriction than we need here.
(iv) We thus see that $Q = R \circ S_2 \circ S_1$ as desired.
(v) Furthermore, by the fact that $v$ is closed, we get $R(v) = v$, as desired.

Case $\text{V-Mono}$:

\[
\begin{array}{c}
\frac{\Gamma \vdash \nu \ e: \tau \\
\text{no other rule matches}}{
\Gamma \vdash \nu \ e: \tau} \\
\end{array}
\]

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

Case $\text{V-Gen}$:

\[
\begin{array}{c}
\pi = \text{ftv}(v) \setminus \text{ftv}(\Gamma) \\
\frac{\Gamma \vdash \nu \ e: \nu}{}
\end{array}
\]

The induction hypothesis gives us $V^*(\Gamma, e) = (S, \nu')$ with $R$ such that $Q = R \circ S$ and $R(\nu') = \nu$. We know $\pi = \text{ftv}(R(\nu')) \setminus \text{ftv}(R(S(\Gamma)))$, and let $\pi' = \text{ftv}(\nu') \setminus \text{ftv}(S(\Gamma))$.
(i) We see that $V^{\text{gen}}(\Gamma, e) = (S, \forall{\pi'}, \nu')$.
(ii) Choose $R$ as from the induction hypothesis.
(iii) The domain restriction is via the induction hypothesis.
(iv) We know $Q = R \circ S$.
We must show $R(\forall{\pi'}, \nu') \leq_{\text{hmv}} \forall{\pi}, v$.
(v) This is direct from Lemma 21

Proof (Proof of principal types for HMV (Theorem 4)). By completeness of $V$ (Theorem 6), we have $\sigma'_0$ such that $\Gamma^{\text{gen}} \ e: \sigma'_0$ and $\sigma'_0 \leq_{\text{hmv}} \sigma$. By completeness of Algorithm $V$ (Lemma 22), we have $V^{\text{gen}}(\Gamma, e) = (S, \sigma'_0)$ with $R$ such that $e = R \circ S$ and $R(\sigma'_0) \leq_{\text{hmv}} \sigma'_0$. Let $\sigma_p = R(\sigma'_0)$.

By the soundness of Algorithm $V$ (Lemma 20), we know that $S(\Gamma)^{\text{gen}} \ e: \sigma'_p$, or equivalently: $S(\Gamma)^{\text{gen}} \ e: S(\sigma_p)$. By substitution, we can substitute through by $R$ to get $\Gamma^{\text{gen}} \ e: \sigma_p$. By soundness of $V$ (Theorem 5), we have $\Gamma^{\text{hmv}} \ e: \sigma_p$. Recall that $\sigma_p \leq_{\text{hmv}} \sigma_0$. But we assumed nothing about $\sigma_0$. Thus, $\sigma_p$ is a principal type for $e$.

Proof (Proof of decidability of System $V$ (Theorem 7)). All that remains is to show that Algorithm $V$ terminates. All cases in Algorithm $V$ 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 $V^{\text{gen}}(\Gamma, e)$ to $V^*(\Gamma, e)$, it, too, cannot loop.
Higher-rank systems: properties of $\leq_{b}$

This section concerns the properties of the first order subsumption, higher-order instantiation relation, $\sigma_1 \leq_{b} \sigma_2$. For reference this relation is repeated in Fig. 12.

\[
\begin{align*}
\sigma_1 & \leq_{b} \sigma_2 \quad \text{Higher-rank instantiation} \\
\tau & \leq_{b} \tau & \text{B\_REFL} \\
\sigma_1 & \leq_{b} \sigma_2 & \text{B\_FUN} \\
\phi_1[\tau/b] & \leq_{b} \phi_2 & \text{B\_INSTS} \\
\forall \pi, b. \phi_1 & \leq_{b} \forall \pi. \phi_2 & \text{B\_INSTG} \\
\end{align*}
\]

Fig. 12. Inner instantiation

Lemma 23 (Monotypes are already instantiated). If $\tau_1 \leq_{b} \tau_2$ then $\tau_1 = \tau_2$.

Proof. By inversion.

Lemma 24 (Substitution for instantiation). If $\sigma_1 \leq_{b} \sigma_2$ then $S(\sigma_1) \leq_{b} S(\sigma_2)$.

Lemma 25 (Reflexivity for $\leq_{b}$). For all $\sigma$, $\sigma \leq_{b} \sigma$.

Lemma 26 (Transitivity for $\leq_{b}$). If $\sigma_1 \leq_{b} \sigma_2$ and $\sigma_2 \leq_{b} \sigma_3$, then $\sigma_1 \leq_{b} \sigma_3$.

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

\[
\begin{align*}
H(\tau) &= 1 \\
H(\sigma_1 \rightarrow \sigma_2) &= 1 + \max(H\sigma_1, H\sigma_2) \\
& \quad \text{when at least one of } \sigma_1 \text{ and } \sigma_2 \\
& \quad \text{is not a monotype} \\
H(\forall \pi. \phi) &= 1 + H(\phi) \\
H(\forall \{\pi\}. \sigma) &= 1 + H(\sigma)
\end{align*}
\]

Note that the $H$ function is stable under substitution (replacing variables by monotypes).
Case $\sigma_2$ is $\tau$, a monotype
In this case, by Lemma 23, we know that $\sigma_3$ must also be $\tau$. So the result holds by assumption.

Case $\sigma_2$ is $v_{21} \rightarrow v_{22}$
By inversion, we know that $\sigma_1$ is $\forall \varpi, \bar{b}, \tau. v_{11} \rightarrow v_{12}$ such that $v_{21} \leq_b v_{11}[\tau/\bar{b}, \tau]$ and $v_{12}[\tau/\bar{b}, \tau] \leq_b v_{12}$.
We also know that $\sigma_3$ is $\forall \{\bar{d}\}. v_{31} \rightarrow v_{32}$, such that $v_{31} \leq_b v_{21}$ and $v_{32} \leq_b v_{32}$.
By induction, we can show $v_{31} \leq_b v_{11}[\tau/\bar{b}, \tau]$ and $v_{12}[\tau/\bar{b}, \tau] \leq_b v_{32}$.
This lets us conclude that $\sigma_1 \leq_b \sigma_2$.

Case $\sigma_2$ is $\forall \varpi, \phi_2$
By inversion, we know that $\sigma_1$ is $\forall \{\bar{b}\}, \bar{b}, \tau. \phi_1$ such that $\phi_1[\tau/\bar{b}, \tau] \leq_b \phi_2$.
We also know that $\sigma_3$ is $\forall \{\bar{d}\}, \bar{d}, \phi_3$ where $\bar{d} = \bar{d}_1, \bar{d}_2$ and $\phi_2[\tau'/\bar{d}_2] \leq_b \phi_3$.
By substitution, we can show that $\phi_3[\tau/\bar{b}, \tau][\tau'/\bar{d}_2] \leq_b \phi_2[\tau'/\bar{d}_2]$.
By induction, we then have $\phi_1[\tau/\bar{b}, \tau][\tau'/\bar{d}_2] \leq_b \phi_3$. We can then derive $\sigma_1 \leq_b \sigma_3$ to conclude.

Case $\sigma_2$ is $\forall \tau. v_1$
By inversion, we know that $\sigma_1$ is $\forall \{\bar{b}\}. v_1$ where $v_1[\tau/\bar{b}] \leq_b v_2$.
We also know that $\sigma_3$ is $\forall \{\bar{d}\}. v_3$, where $v_2[\tau'/\bar{d}] \leq_b v_3$.
By substitution, we can show that $v_1[\tau/\bar{b}][\tau'/\bar{d}] \leq_b v_2[\tau'/\bar{d}]$. Rewrite this as $v_1[\tau[\tau'/\bar{d}]/\bar{b}] \leq_b v_2[\tau'/\bar{d}]$.
By induction, we have $v_1[\tau[\tau'/\bar{d}]/\bar{b}] \leq_b v_3$.
Therefore we can conclude $\forall \{\bar{b}\}. v_1 \leq_b v_3$.

G Higher-rank systems: properties of DSK System B

This section considers the Higher-Rank type systems with deep-skolemization, described in Section 6.0

G.1 Properties of Prenex conversion

Lemma 27 (Instantiation and Prenex). If $v \leq_b v'$ and $\text{prenex}(v) = \forall \varpi. \rho$ and $\text{prenex}(v') = \forall \bar{b}. \rho'$, then $\forall \varpi. \rho \leq_b \forall \bar{b}. \rho'$ and $\bar{b} \subseteq \varpi$.

Proof. Proof is by induction on $v \leq_b v'$.

Case $\text{B}_\text{INSTS}$:

$$\frac{\phi_1[\tau/\bar{b}] \leq_b \phi_2}{\forall \varpi, \bar{b}. \phi_1 \leq_b \forall \varpi. \phi_2} \text{B}_\text{INSTS}$$

Say that $\text{prenex}(v) = \forall \varpi, \bar{b}, \tau. \rho'_1$ where $\text{prenex}(\phi) = \forall \varpi. \rho'_1$.
This means that $\text{prenex}(\phi_1[\tau/\bar{b}]) = \forall \varpi. \rho'_1[\tau/\bar{b}]$ as $\tau$ do not contain quantifiers.
Say also that $\text{prenex}(v') = \forall \varpi, \bar{d}. \rho'_2$ where $\text{prenex}(\phi_2) = \forall \bar{d}. \rho'_2$.
By induction, we have that $\rho'_1[\tau/\bar{b}] \leq_b \rho'_2$ where $\bar{d} \subseteq \tau$.
Therefore we can conclude by $\text{B}_\text{INSTS}$ that $\forall \varpi. \rho'_1 \leq_b \forall \bar{b}. \rho'_2$.
Case B\_FUN:

\[ \frac{v_3 \leq_b v_1 \quad v_2 \leq_b v_4}{v_1 \rightarrow v_2 \leq_b v_3 \rightarrow v_4} \text{B\_FUN} \]

Note that \( \text{prenex}(v_1 \rightarrow v_2) = \forall \pi. v_1 \rightarrow \rho_2 \) where \( \forall \pi. \rho_2 = \text{prenex}(v_2) \) and the \( \pi \) are not free in \( v_1 \). \( \text{prenex}(v_3 \rightarrow v_4) = \forall \overline{b}. v_3 \rightarrow \rho_4 \) where \( \forall \overline{b}. \rho_4 = \text{prenex}(v_4) \). So by induction, we have that \( \forall \pi. \rho_2 \leq_b \forall \overline{b}. \rho_4 \) and \( \overline{b} \subseteq \pi \).

By inversion, we have that \( \rho_2[\overline{\pi}/\overline{\pi}] \leq_b \rho_4 \). From this we can derive \( v_1 \rightarrow \rho_2[\overline{\pi}/\overline{\pi}] \leq_b v_3 \rightarrow \rho_4 \) and then \( \forall \pi. v_1 \rightarrow \rho_2 \leq_b \forall \overline{b}. v_3 \rightarrow \rho_4 \).

Case B\_REFL: trivial.

Lemma 28 (Prenex instantiates). If \( \text{prenex}(v) = \forall \pi. \rho \) then \( v \leq_b \rho \).

Proof. Proof is by induction on \( v \). If \( v \) is a monotype, we are done. If \( v = v_1 \rightarrow v_2 \), then \( \text{prenex}(v) = \forall \pi. v_1 \rightarrow \rho_2 \) where \( \forall \pi. \rho_2 = \text{prenex}(v_2) \). By induction, we know that \( v_2 \leq_b \rho_2 \), so by B\_FUN, we have \( v_1 \rightarrow v_2 \leq_b v_1 \rightarrow \rho_2 \).

If \( v \) is a specified polytype, of the form \( \forall \pi. \phi \), then \( \text{prenex}(v) = \forall \pi. \overline{b}, \rho'_2 \) where \( \forall \pi. \phi = \forall \overline{b}, \rho'_2 \). By induction, we know that \( \rho'_1 \leq_b \rho'_2 \). We can then show that \( \forall \pi. \overline{b}, \rho'_1 \leq_b \rho'_2 \) by B\_INSTS (and instantiating variables with themselves.)

G.2 Properties of DSK subsumption

Lemma 29 (Substitution for higher-rank subsumption). If \( v_1 \leq_{dsk} v_2 \) then \( S(v_1) \leq_{dsk} S(v_2) \). If \( \sigma_1 \leq_{dsk} v_2 \) then \( S(\sigma_1) \leq_{dsk} S(v_2) \).

See Vytiniotis et al. [27] Lemma 2.7.

Lemma 30 (Reflexivity for \( \leq_{dsk} \)). For all \( v, v \leq_{dsk} v \).

Proof. Vytiniotis et al., [27] Lemma 2.21.

Lemma 31 (Transitivity for \( \leq_{dsk} \)). If \( v_1 \leq_{dsk} v_2 \) and \( v_2 \leq_{dsk} v_3 \), then \( v_1 \leq_{dsk} v_3 \).

Proof. Vytiniotis et al., [27] Lemma 2.22.

Lemma 32 (Single skol admissible). If \( \sigma_1 \leq_{dsk} v_2 \) then \( \sigma_1 \leq_{dsk} \forall c. v_2 \) (when \( c \) are not free in \( \sigma_1 \)).

Proof. Proof is by induction on \( \sigma_1 \leq_{dsk} v_2 \).

Case DSK\_REFL: Direct by DSK\_INSTS.

Case DSK\_FUN:

\[ \frac{v_3 \leq_{dsk} v_1 \quad v_2 \leq_{dsk} \rho_4}{v_1 \rightarrow v_2 \leq_{dsk} v_3 \rightarrow \rho_4} \text{DSK\_FUN} \]

Say \( \text{prenex}(v_4) = \forall \overline{b}. \rho \). By inversion, we know that \( v_1 \rightarrow v_2 \leq_{dsk} v_3 \rightarrow \rho \).

We also know that \( \text{prenex}(\forall c. v_3 \rightarrow v_4) = \forall c. \overline{b}. \overline{c} v_3 \rightarrow \rho \), so we can conclude with DSK\_INST.
Case DSK _INST:

\[
\begin{align*}
\text{prenex}(v_2) &= \forall \tau. \rho_2 \\
\phi_1[\tau/\pi][\tau'/\beta] &\leq_{\text{DSK}} \rho_2 \\
\forall \{\pi\}, \beta. \phi_1 &\leq_{\text{DSK}} v_2 \\
\end{align*}
\]

We note that \(\text{prenex}(\forall c. v_2) = \forall c, \beta, \rho_2\), so we can just apply DSK _INST.  

Case SB _INST:

\[
\begin{align*}
\text{prenex}(v_2) &= \forall \tau. \rho_2 \\
\phi_1[\tau/\pi][\tau'/\beta] &\leq_{\text{DSK}} \rho_2 \\
\forall \{\pi\}, \beta. \phi_1 &\leq_{\text{DSK}} v_2 \\
\end{align*}
\]

Similar reasoning to DSK _INST.

Lemma 33 (Monotypes are instantiations). If \(\sigma \leq_{\text{DSK}} \tau\) then \(\sigma \leq_{\text{b}} \tau\).

Proof. Proof is by induction on \(\sigma \leq_{\text{DSK}} \tau\). In each case the result holds directly by induction.

Lemma 34 (DSK and prenex). For all \(v\), we have \(v \leq_{\text{DSK}} \text{prenex}(v)\).

Proof. Proof is by induction on \(v\).

Lemma 35 (DSK Subsumption contains OL Subsumption). If \(v_1 \leq_{\text{ol}} v_2\) then \(v_1 \leq_{\text{DSK}} v_2\).

Proof. Proof is by induction on the derivation.

Case OL _B _AREFL: Trivial.

Case OL _B _AFUN: By induction.

Case OL _B _AINSTS:

\[
\begin{align*}
\phi_1[\tau/\pi] &\leq_{\text{ol}} \phi_2 \\
\beta \notin \text{ftv}(\forall \pi. \phi_1) \\
\forall \pi. \phi_1 &\leq_{\text{ol}} \forall \beta. \phi_2 \\
\end{align*}
\]

By induction we know that \(\phi_1[\tau/\beta] \leq_{\text{DSK}} \phi_2\). We want to show that \(\forall \pi. \phi_1 \leq_{\text{DSK}} \forall \beta. \phi_2\). Say \(\text{prenex}(\phi_2) = \forall \tau. \rho_2\), then by DSK _INST it suffices to show that \(\phi_1[\tau'/\beta] \leq_{\text{DSK}} \rho_2\). Note that as \(\forall \tau. \rho_2 \leq_{\text{DSK}} \rho_2\), by transitivity, we can reduce this to showing \(\phi_1[\tau'/\beta] \leq_{\text{DSK}} \text{prenex}(\phi_2)\).

We can finish, again by transitivity, by showing via Lemma 34, that \(\phi_2 \leq_{\text{DSK}} \text{prenex}(\phi_2)\).

Corollary 36 (DSK Subsumption contains Instantiation). If \(v_1 \leq_{\text{b}} v_2\) then \(v_1 \leq_{\text{DSK}} v_2\).

Proof. See above and Lemma 48.

Lemma 37 (Transitivity of Higher-Rank subsumption I). If \(\sigma_1 \leq_{\text{b}} \sigma_2\) and \(\sigma_2 \leq_{\text{DSK}} \sigma_3\), then \(\sigma_1 \leq_{\text{DSK}} \sigma_3\).

Proof. Follows from Lemmas 31 and 36.

Lemma 38 (Transitivity of Higher-Rank subsumption II). If \(\sigma_1 \leq_{\text{DSK}} v_2\) and \(v_2 \leq_{\text{b}} v_3\), then \(\sigma_1 \leq_{\text{DSK}} v_3\).

Proof. Follows from Lemmas 31 and 36.
G.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 in more restrictive forms.

**Lemma 39 (Substitution for System B).** Assume that the domain of $S$ is disjoint from the variables of $\Gamma$ and from the free type variables of $e$.

1. If $\Gamma \vdash b : e \Rightarrow \sigma$ then $\Gamma \vdash b : e \Rightarrow S(\sigma)$
2. If $\Gamma \vdash b : e \Leftarrow \upsilon$ then $\Gamma \vdash b : e \Leftarrow S(\upsilon)$

**Lemma 40 (Substitution for System SB).** Assume that the domain of $S$ is disjoint from the variables of $\Gamma$ and from the free type variables of $e$.

1. If $\Gamma \vdash sb : e \Rightarrow \phi$ then $\Gamma \vdash sb : e \Rightarrow S(\phi)$.
2. If $\Gamma \vdash \ast sb : e \Rightarrow \upsilon$ then $\Gamma \vdash \ast sb : e \Rightarrow S(\upsilon)$.
3. If $\Gamma \vdash \text{gen} sb : e \Rightarrow \sigma$ then $\Gamma \vdash \text{gen} sb : e \Rightarrow S(\sigma)$.
4. If $\Gamma \vdash \ast sb : e \Leftarrow \upsilon$ then $\Gamma \vdash \ast sb : e \Leftarrow S(\upsilon)$.
5. If $\Gamma \vdash sb : e \Leftarrow \rho$ then $\Gamma \vdash sb : e \Leftarrow S(\rho)$.

G.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 41 (Prenex System B).** If $\Gamma \vdash b : e \Leftarrow \rho$ and prenex($\phi$) = $\forall \pi. \rho$ then $\Gamma \vdash b : e \Leftarrow \phi$.

**Proof.** Case $\text{B\_DAbs}$

$$
\frac{\Gamma, x : v_1 \vdash b : e \Leftarrow v_2 \quad \text{B\_DAbs}}{\Gamma \vdash \lambda x. e \Leftarrow v_1 \rightarrow v_2}
$$

In this case, because we are pushing in $\rho$, we have $v_2 = \rho_2$. We also know that prenex($\phi$) = $\forall \pi. v_1 \rightarrow \rho_2$, so $\phi$ must be of the form $v_1 \rightarrow v_2$.

Write $v_2$ as $\forall \pi_1, \phi_2$. We also know that prenex($v_2$) = $\forall \pi_2, \rho_2$ where prenex($\phi_2$) = $\forall \pi_2, \rho_2$.

By induction, we have $\Gamma, x : v_1 \vdash b : e \Leftarrow \phi_2$. By (repeated) use of $\text{B\_SKOL}$, we have $\Gamma, x : v_1 \vdash e \Leftarrow \forall \pi. \phi_2$. By $\text{B\_DAbs}$, we can then conclude $\Gamma \vdash \lambda x. e \Leftarrow v_1 \rightarrow \forall \pi, \phi_2$, which is what we wanted to show.

Case $\text{B\_DLet}$

$$
\frac{\Gamma \vdash b : e_1 \Rightarrow \sigma \quad \Gamma, x : \sigma_1 \vdash e_2 \Leftarrow \upsilon \quad \text{B\_DLet}}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Leftarrow \upsilon}
$$

This case holds by induction.
Case B_Skol This case is impossible, as the conclusion does not have the form $\phi$.

Case B_Infer

\[ \frac{\Gamma \vdash b \sigma_1 \sigma_1 \leq_{dsk} v_2}{\Gamma \vdash b e \leftarrow v_2} \text{B_Infer} \]

Here we know that $v_2$ is $\rho$. By inversion of $\sigma \leq_{dsk} \rho$, we have $\sigma = \forall\{\pi_1, \overline{b}_1, \phi_2\}$ where $\text{prenex}(\phi_2) = \forall \overline{\pi}, \rho$ and $\rho_2[\pi/\pi_1][\overline{\pi}/\overline{b}_1] \leq_{dsk} \rho$.

However, as we also have $\text{prenex}(\phi) = \forall \overline{\alpha}, \rho$, we can use the same information to conclude, $\sigma \leq_{dsk} \phi$, and then use B_Infer to show that $\Gamma \vdash b e \leftarrow \phi$.

Proof Soundness of System SB Lemma 8 states:

1. If $\Gamma \vdash_{sb} e \Rightarrow \phi$ then $\Gamma \vdash_{b} e \Rightarrow \phi$.
2. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_{b} e \Rightarrow v$.
3. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then $\Gamma \vdash_{b} e \Rightarrow \sigma$.
4. If $\Gamma \vdash_{sb} e \Leftarrow v$ then $\Gamma \vdash_{b} e \Leftarrow v$.
5. If $\Gamma \vdash_{sb} e \Leftarrow \rho$ then $\Gamma \vdash_{b} e \Leftarrow \rho$.

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:

\[ \frac{\Gamma \vdash v \quad v = \forall \overline{\pi}, \overline{b}, \phi \quad \Gamma, \overline{\pi} \vdash_{sb} e \Leftarrow \phi \quad \overline{\pi}, \overline{b} \not\in \text{ftv}(\Gamma)}{\Gamma \vdash_{sb} (\Lambda \overline{\pi}. e : v) \Rightarrow v} \text{SB_Annot} \]

By induction we know that $\Gamma, \overline{\pi} \vdash_{b} e \Leftarrow \phi$. By B_Annot, we can conclude $\Gamma \vdash_{b} (\Lambda \overline{\pi}. e : v) \Rightarrow v$.

Case SB_Infer:

\[ \frac{\Gamma \vdash_{sb} e \Rightarrow v_1 \quad v_1 \leq_{dsk} \rho_2 \quad \text{no other rule matches}}{\Gamma \vdash_{sb} e \Leftarrow \rho_2} \text{SB_Infer} \]

By induction, we know that $\Gamma \vdash_{b} e \Rightarrow v_1$. We would like to show that $\Gamma \vdash_{b} e \Leftarrow v_1$. This follows immediately by B_Infer.

Case SB_DeepSkol

\[ \frac{\text{prenex}(v) = \forall \overline{\pi}, \rho \quad \overline{\pi} \not\in \text{ftv}(\Gamma)}{\Gamma \vdash_{sb} e \Leftarrow \rho} \text{SB_DeepSkol} \]

By induction, we have $\Gamma \vdash_{b} e \Leftarrow \rho$. Note that if $\text{prenex}(v) = \forall \overline{\pi}, \rho$ and $v$ is of the form $\forall \overline{\pi}', \phi$, then, by definition $\text{prenex}(\phi) = \forall \overline{\pi}'', \rho$ and $\overline{\pi} = \overline{\pi}', \overline{\pi}''$. By the prenex lemma 41 we know that $\Gamma \vdash_{b} e \Leftarrow \phi$. We can then use multiple applications of rule B_Skol to conclude $\Gamma \vdash_{b} e \Leftarrow \forall \overline{\pi}', \phi$. 

G.5 Completeness of syntax-directed system

Lemma 42 (Context Generalization). Suppose $\Gamma' \subseteq_b \Gamma$

1. If $\Gamma|_{sb} e \Rightarrow \phi$ then there exists $\phi' \subseteq_b \phi$ such that $\Gamma'|_{sb} e \Rightarrow \phi'$.
2. If $\Gamma|_{sb} e \Rightarrow v$ then there exists $v' \subseteq_b v$ such that $\Gamma'|_{sb} e \Rightarrow v'$.
3. If $\Gamma|_{sb} e \Rightarrow \sigma$ then there exists $\sigma' \subseteq_b \sigma$ such that $\Gamma'|_{sb} e \Rightarrow \sigma'$.
4. If $\Gamma|_{sb} e \Leftarrow v$ and $v \subseteq_b v'$ then $\Gamma'|_{sb} e \Leftarrow v'$.
5. If $\Gamma|_{sb} e \Leftarrow \rho$ and $\rho \subseteq_b \rho'$ then $\Gamma'|_{sb} e \Leftarrow \rho$.

Proof. Proof is by induction on derivations.

Case SB_Abs:

$$\frac{\Gamma, x : \tau \ 
\Gamma|_{sb} \lambda x. e \Rightarrow \tau \rightarrow v \ 
\text{SB_Abs}}{\Gamma|_{sb} \ x \ 
\text{SB_Var}}$$

By induction, we know that $\Gamma', x : \tau \ 
\Gamma|_{sb} \lambda x. e \Rightarrow \tau \rightarrow v'$ for $v' \subseteq_b v$. Therefore, $\Gamma'|_{sb} \lambda x. e \Rightarrow \tau \rightarrow v'$ and, by SB_Fun, $\tau \rightarrow v' \subseteq_b \tau \rightarrow v$.

Case SB_Int:

$$\frac{\Gamma|_{sb} n \Rightarrow \text{Int} \ 
\text{SB_Int}}{}$$

Trivial.

Case SB_Insts:

$$\frac{\Gamma|_{sb} e \Rightarrow \forall \overline{a}, \phi \ 
\text{no other rule matches}}{\Gamma|_{sb} e \Rightarrow \phi[\overline{a}] \ 
\text{SB_Insts}}$$

By induction, we know that $\Gamma|_{sb} e \Rightarrow v'$ where $v' \subseteq_b \forall \overline{a}, \phi$. By inversion, we know that $v'$ must be of the form $\forall \overline{a}, \phi'$ where $\phi'[\overline{a}] \subseteq_b \phi$. By SB_Insts, we can conclude $\Gamma|_{sb} e \Rightarrow (\phi'[\overline{a}])[\overline{a}]$. We also need to show that $(\phi'[\overline{a}])[\overline{a}] \subseteq_b \phi[\overline{a}]$, which follows by substitution (Lemma 24).

Case SB_Var:

$$\frac{x : \forall \{\overline{a}\}, v \in \Gamma \ 
\Gamma|_{sb} x \Rightarrow v[\overline{a}] \ 
\text{SB_Var}}{}$$

We know that $x : \sigma \in \Gamma$, where $\sigma \subseteq_b \forall \{\overline{a}\}, v$. So by inversion, $\sigma$ must be $\forall \{\overline{a}\}, v'$ such that $v'[\overline{a}] \subseteq_b v$. Therefore, by substitution lemma 24, $v'[\overline{a}] \subseteq_b v[\overline{a}]$. As we know that the $\overline{a}$ are not free in $v'$, we can rewrite the left hand side as: $v'[\overline{a}]$, and choose those types in the use of SB_Var.

Case SB_App:

$$\frac{\Gamma|_{sb} e_1 \Rightarrow v_1 \rightarrow v_2 \ 
\Gamma|_{sb} e_2 \Leftarrow v_1 \ 
\text{SB_App}}{\Gamma|_{sb} e_1 \Rightarrow v_2 \ 
\Gamma|_{sb} e_2 \Leftarrow v_1'}$$

By induction we have $\Gamma'|_{sb} e_1 \Rightarrow \phi$ such that $\phi \subseteq_b v_1 \rightarrow v_2$. By inversion, this means that $\phi$ must be of the form $v_1' \rightarrow v_2'$ where $v_1 \subseteq_b v_1'$ and $v_2 \subseteq_b v_2$.

By induction, we have $\Gamma'|_{sb} e_2 \Leftarrow v_1'$. So we can conclude by SB_App.
Case SB_Let:

\[
\frac{\Gamma \vdash^\text{gen} e_1 \Rightarrow \sigma_1 \quad \Gamma, x: \sigma_1 \vdash^\text{sb} e_2 \Rightarrow \nu_2}{\Gamma \vdash^\text{lsb} \text{let } x = e_1 \text{ in } e_2 \Rightarrow \nu_2} \text{SB_Let}
\]

By induction \(\Gamma' \vdash^\text{gen} e_1 \Rightarrow \sigma'_1 \) for \(\sigma'_1 \leq_b \sigma_1\). By induction (again), \(\Gamma', x: \sigma'_1 \vdash^\text{lsb} e_2 \Rightarrow \nu'_2\) for \(\nu'_2 \leq_b \nu_2\). So we can conclude by SB_Let.

Case SB_TApp:

\[
\frac{\Gamma \vdash \tau \quad \Gamma \vdash^\text{sb} e \Rightarrow \forall a. \nu \quad \Gamma \vdash^\text{lsb} (\lambda x.e) : \nu}{\Gamma \vdash^\text{lsb} e \Rightarrow \nu[\tau/a]} \text{SB_TApp}
\]

By induction \(\Gamma' \vdash^\text{sb} e \Rightarrow \nu'\) where \(\nu' \leq_b \forall a. \nu\). By inversion, \(\nu'\) is of the form \(\forall a. \nu_1\) where \(\nu_1 \leq_b \nu\). By substitution, \(\nu_1[\tau/a] \leq_b \nu[\tau/a]\).

Case SB_Annot:

\[
\frac{\Gamma \vdash v \quad v = \forall \pi, b, \phi \quad \Gamma, \pi \vdash^\text{lsb} e \Leftarrow \phi \quad \pi, b \notin \text{ftv}(\Gamma)}{\Gamma \vdash^\text{lsb} \Lambda \pi.e : v} \text{SB_Annot}
\]

By induction, we have \(\Gamma' \vdash^\text{lsb} e \Leftarrow \phi\), so we can use SB_Annot to conclude \(\Gamma' \vdash^\text{lsb} e \Rightarrow v\).

Case SB_Phi:

\[
\frac{\Gamma \vdash^\text{lsb} e \Rightarrow \phi \quad \text{no other rule matches}}{\Gamma \vdash^\text{lsb} e \Rightarrow \phi} \text{SB_Phi}
\]

Holds directly by induction.

Case SB_Gen:

\[
\frac{\Gamma \vdash^\text{lsb} e \Rightarrow v \quad \pi = \text{ftv}(v) \setminus \text{ftv}(\Gamma)}{\Gamma \vdash^\text{gen} \Rightarrow \forall \{\pi\}. v} \text{SB_Gen}
\]

By induction, we have \(\Gamma' \vdash^\text{lsb} e \Rightarrow v'\) for \(v' \leq_b v\). Let \(\overline{b} = \text{ftv}(v') \setminus \text{ftv}(\Gamma)\). We want to prove that \(\forall \{\overline{b}\}. v' \leq_b \forall \{\pi\}. v\). This holds by B_INSTG, choosing \(\tau = \overline{\pi}\).

Case SB_DAbs:

\[
\frac{\Gamma, x: \nu_1 \vdash^\text{lsb} e \Leftarrow \rho_2}{\Gamma \vdash^\text{lsb} \lambda x. e \Leftarrow v_1 \Rightarrow \rho_2} \text{SB_DAbs}
\]

Let \(\Gamma' \leq_b \Gamma\) and \(v_1 \Rightarrow \rho_2 \leq_b v'\) be arbitrary. By inversion, we know that \(v'\) is \(v'_1 \Rightarrow v'_2\) where \(v'_1 \leq_b v_1\) and \(\rho_2 \leq_b v'_2\).

By inversion, we know that \(v'_2\) must be of the form \(\rho'_2\), as \(\leq_b\) cannot introduce specified quantifiers.

By induction, we know that \(\Gamma', x: v'_1 \vdash^\text{lsb} e \Leftarrow \rho'_2\).

So we can conclude by SB_DAbs.
Case **SB_INFER**:

\[
\Gamma \vdash_{\text{sb}} e \Rightarrow v_1 \quad v_1 \leq_{\text{dsk}} \rho_2 \\
\text{no other rule matches}
\]

\[
\Gamma \vdash_{\text{sb}} e \Leftarrow \rho_2
\]

\text{SB_INFER}

Let \( \Gamma' \leq_b \Gamma \) and \( \rho_2 \leq_b \rho' \) be arbitrary. By induction, we know that \( \Gamma' \vdash_{\text{sb}} e \Rightarrow v' \) for \( v' \leq_b v_1 \). We want to show that \( v' \leq_{\text{dsk}} \rho' \). This holds by lemmas 37 and 38.

Case **SB_DLET**:

\[
\Gamma \vdash_{\text{gen}} e_1 \Rightarrow \sigma_1 \\
\Gamma, x: \sigma_1 \vdash_{\text{sb}} e_2 \Leftarrow \rho_2
\]

\text{SB_DLET}

Let \( \Gamma' \leq_b \Gamma \) and \( \rho_2 \leq_b \rho' \) be arbitrary. By induction, we have \( \Gamma' \vdash_{\text{gen}} e_1 \Rightarrow \sigma_1' \) for \( \sigma_1' \leq_b \sigma_1 \). By induction (again), \( \Gamma', x: \sigma_1' \vdash_{\text{sb}} e_2 \Leftarrow \rho' \). So we can conclude by \text{SB_DLET}.

Case **SB_DeePskol**

\[
\text{prenex}(v) = \forall \pi, \rho \\
\pi \notin \text{ftv}(\Gamma) \\
\Gamma \vdash_{\text{sb}} e \Leftarrow \rho
\]

\text{SB_DeePskol}

Let \( \Gamma' \leq_b \Gamma \) and \( v \leq_b v' \) be arbitrary. Let \( \text{prenex}(v') = \forall \pi', \rho' \). By Lemma 28 we know that \( \rho \leq_b \rho' \). So by induction, \( \Gamma' \vdash_{\text{sb}} e \Leftarrow \rho' \). Therefore, we can use \text{SB_DeePskol} to conclude.

**Proof of Completeness of System SB** Lemma 9 states:

1. If \( \Gamma \vdash_{\text{sb}} e \Rightarrow \sigma \) then \( \Gamma \vdash_{\text{gen}} e \Rightarrow \sigma' \) where \( \sigma' \leq_b \sigma \).

2. If \( \Gamma \vdash_{\text{sb}} e \Leftarrow v \) then \( \Gamma \vdash_{\text{sb}} e \Leftarrow v \).

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

Case **B_VAR**

\[
x: \sigma \in \Gamma \\
\Gamma \vdash_{\text{b}} x \Rightarrow \sigma
\]

\text{B_VAR}

Suppose \( \sigma \) is for the form \( \forall \{\pi\}. v \). Then we use \( \pi \) to instantiate the variables \( \pi \). We know these \( \pi \) are not free in \( \Gamma \) by the Barendregt convention. It may be the case that generalization quantifies over more variables, i.e. \( \pi \subseteq \pi' = \text{ftv}(v) \setminus \text{ftv}(\Gamma) \), leading to a more general result type. However, that is permitted by the statement of the theorem.
Case \textbf{B\_Abs}

\[
\frac{\Gamma, x : \tau \triangleright e \Rightarrow v}{\Gamma \triangleright \lambda x. e \Rightarrow \tau \rightarrow v} \quad \text{B\_Abs}
\]

The induction hypothesis gives us \(\Gamma, x : \tau \triangleright e \Rightarrow \forall \{\pi\}, v'\) \(\forall\{\pi\}, v' \leq_b v\). Inverting \(\triangleright^{gen}\) gives us \(\Gamma, x : \tau \triangleright e \Rightarrow v'\) and inverting \(\leq_b\) gives us \(v'[[\tau/\pi]] \leq_b v\). We can then substitute and use \textbf{SB\_Abs} to get \(\Gamma \triangleright \lambda x. e \Rightarrow \tau \rightarrow v'[[\tau/\pi]]\). Generalizing, we get \(\Gamma \triangleright^{gen} \lambda x. e \Rightarrow \forall\{\pi, \pi'\}, \tau \rightarrow v'[[\tau/\pi]]\) where the new variables \(\pi'\) come from generalizing \(\tau\) and the \(\pi'\). We are done because \((\tau \rightarrow v'[[\tau/\pi]] \leq_b \tau \rightarrow v)\) and so \(\forall\{\pi, \pi'\}, \tau \rightarrow v' \leq_b \tau \rightarrow v\).

Case \textbf{B\_App}

\[
\frac{\Gamma \triangleright e_1 \Rightarrow v_1 \rightarrow v_2 \quad \Gamma \triangleright e_2 \Leftarrow v_1}{\Gamma \triangleright e_1 e_2 \Leftarrow v_2} \quad \text{B\_App}
\]

By induction we have \(\Gamma \triangleright^{gen}_{sb} e_1 \Rightarrow \sigma\) for some \(\leq_b v_1 \rightarrow v_2\).
So by inversion \(\sigma = \forall\{\pi\}, \tilde{b}, v_1' \rightarrow v_2'\), where \((v_1' \rightarrow v_2')(\tau/\pi)[\tau'/\tilde{b}] \leq_b v_1 \rightarrow v_2\) and \(v_1 \leq_b v_1'(\tau/\pi)[\tau'/\tilde{b}]\).
Also by inversion of \(\triangleright^{gen}_{sb}\) we know that \(\Gamma \triangleright^{gen}_{sb} e_1 \Rightarrow \forall\tilde{b}, v_1' \rightarrow v_2' \) and that \(\tilde{b}\) are not free in \(\Gamma\) or \(e_1\).
By substitution, we have \(\Gamma \triangleright^{gen}_{sb} e_1 \Rightarrow \forall\tilde{b}, (v_1' \rightarrow v_2')(\tau/\pi)[\tau'/\tilde{b}]\).
And by \textbf{SB\_InstS}, we have \(\Gamma \triangleright^{gen}_{sb} e_1 \Rightarrow (v_1' \rightarrow v_2')(\tau/\pi)[\tau'/\tilde{b}]\).
By induction we have \(\Gamma \triangleright^{gen}_{sb} e_2 \Leftarrow v_1'\) and by lemma \(\triangleright^{gen}_{sb}\), we know that \(\Gamma \triangleright^{gen}_{sb} e_2 \Leftarrow v_1'(\tau/\pi)[\tau'/\tilde{b}]\).
So we can conclude \(\Gamma \triangleright^{gen}_{sb} e_1 e_2 \Leftarrow v_2'[\tau/\pi][\tau'/\tilde{b}]\).

Case \textbf{B\_Let}

\[
\frac{\Gamma \triangleright e_1 \Rightarrow \sigma_1 \quad \Gamma, x : \sigma_1 \triangleright e_2 \Rightarrow \sigma}{\Gamma \triangleright \text{let}\ x = e_1 \text{ in } e_2 \Rightarrow \sigma} \quad \text{B\_Let}
\]

The induction hypothesis gives us \(\Gamma \triangleright^{gen}_{sb} e_1 \Rightarrow \sigma_1'\) with \(\sigma_1' \leq_b \sigma_1\). The induction hypothesis also gives us \(\Gamma, x : \sigma_1 \triangleright^{gen}_{sb} e_2 \Rightarrow \sigma_2\) with \(\sigma_2' \leq_b \sigma_2\). Use Lemma \(\triangleright^{gen}_{sb}\) to get \(\Gamma, x : \sigma_1' \triangleright^{gen}_{sb} e_2 \Rightarrow \sigma_2'\) where \(\sigma_2' \leq_b \sigma_2\).
Let \(\sigma_2'' \equiv \forall\{\tilde{b}\}, v\) where \(\tilde{b} = \text{f}t\{\{\tau/\tilde{a}\}\} / \text{f}t\{\Gamma\}\). Inverting \(\triangleright^{gen}_{sb}\) gives us \(\Gamma, x : \sigma_1' \triangleright^{gen}_{sb} e_2 \Rightarrow v\). We then use \textbf{SB\_Let} to get \(\Gamma \triangleright^{gen}_{sb} \text{let}\ x = e_1 \text{ in } e_2 \Rightarrow v\). Generalizing gives us \(\Gamma \triangleright^{gen}_{sb} \text{let}\ x = e_1 \text{ in } e_2 \Rightarrow \forall\{\tilde{b}\}, v\).
Transitivity of \(\leq_b\) (Lemma \(\triangleright^{gen}_{sb}\)) gives us \(\forall\{\tilde{b}\}, v \leq_{hmv} \sigma_2\).

Case \textbf{B\_Int}

\[
\frac{\Gamma \triangleright n \Rightarrow \text{Int}}{\Gamma \triangleright n \Rightarrow \text{Int}} \quad \text{B\_Int}
\]

Trivial.

Case \textbf{B\_TApp}

\[
\frac{\Gamma \triangleright \tau}{\Gamma \triangleright \text{e} \Rightarrow \forall a, v} \quad \frac{\Gamma \triangleright e \Rightarrow \forall a, v'}{\Gamma \triangleright e \otimes \tau \Rightarrow v[[\tau/a]]} \quad \text{B\_TApp}
\]

The induction hypothesis (after inverting \(\triangleright^{gen}_{sb}\)) gives us \(\Gamma \triangleright^{gen}_{sb} e \Rightarrow \forall a, v'\), where \(\tilde{b} = \text{f}t\{\forall a, v'\} / \text{f}t\{\Gamma\}\) and \(v'[[\tau/\tilde{b}]] \leq_b v\). Applying \textbf{SB\_TApp}
The induction hypothesis gives us $\Gamma\vdash_{sb} e \Rightarrow v'[\tau/a]$, and $\text{SB\_Gen}$ gives us $\Gamma\vdash_{sb} e \Rightarrow \forall\{\tau\}.v'[\tau/a]$ where $\overline{\tau} = \text{ftv}(v'[\tau/a]) \setminus \text{ftv}(\Gamma)$. We want to show that $\forall\{\tau\}.v'[\tau/a] \leq_b v[\tau/a]$, which follows when there is some $\tau'$, such that $v'[\tau/a][\overline{\tau}/\tau] \leq_b v[\tau/a]$.

This is equivalent to exchanging the substitution, i.e. finding a $\tau'$ such that $v'[\tau'/\tau][\tau/a] \leq_b v[\tau/a]$.

By Substitution (Lemma 24), we have $v'[\tau'/\tau][\tau/a] \leq_b v[\tau/a]$. We also know that the $\overline{b}$ are a subset of the $\overline{\tau}$. So we can choose $\tau'$ to be $\tau$ for the $\overline{b}$, and the remaining $\tau$ elsewhere, and we are done.

**Case B\_ANNOT**

$$\Gamma \vdash v \quad v = \forall\overline{\tau}.\overline{b}, \phi$$

$$\frac{\Gamma, \pi \vdash_b e \Leftarrow \phi \quad \overline{\tau}, \overline{b} \not\in \text{ftv}(\Gamma)}{\Gamma \vdash_b (\Lambda x : v) \Rightarrow v} \text{ B\_ANNOT}$$

By induction, we have $\Gamma, \pi \vdash_{sb} e \Leftarrow \phi$. We can conclude by $\text{SB\_ANNOT}$.

**Case B\_GEN**

$$\frac{\Gamma \vdash_{sb} e \Rightarrow \sigma \quad a \not\in \text{ftv}(\Gamma)}{\Gamma \vdash_{sb} e \Rightarrow \forall\{a\}.\sigma} \text{ B\_GEN}$$

The induction hypothesis gives us $\Gamma\vdash_{sb} e \Rightarrow \sigma'$ where $\sigma' \leq_b \sigma$. We know $\sigma' \leq_b \forall\{a\}.\sigma$. In other words, if $\sigma' = \forall\{\overline{b}\}.v_1$ and $\sigma = \forall\{\tau\}.v_2$, we have some $\overline{\tau}$ such that $v_1[\overline{\tau}/\overline{b}] = v_2$. By the definition of $\leq_b$ we can use these same $\overline{\tau}$ to show that $\sigma' \leq_b \forall\{a, \tau\}.v_2$.

**Case B\_SUB**

$$\frac{\Gamma \vdash_{sb} e \Rightarrow \sigma_1 \quad \sigma_1 \leq_b \sigma_2}{\Gamma \vdash_{sb} e \Rightarrow \sigma_2} \text{ B\_SUB}$$

The induction hypothesis gives us $\Gamma\vdash_{sb} e \Rightarrow \sigma'$ where $\sigma' \leq_b \sigma_1$. By transitivity of $\leq_b$, we are done.

**Case B\_DABS**

$$\frac{\Gamma, x:v_1 \vdash_{sb} e \Leftarrow v_2}{\Gamma \vdash_{sb} \lambda x. e \Leftarrow v_1 \rightarrow v_2} \text{ B\_DABS}$$

By induction, we have that $\Gamma, x:v_1 \vdash_{sb} e \Leftarrow v_2$.

By inversion, we have $\text{prenex}(v_2) = \forall\overline{\tau}.\rho$ and $\Gamma, x:v_1 \vdash_{sb} e \Leftarrow \rho$. Therefore by $\text{SB\_DABS}$, we can conclude $\Gamma \vdash_{sb} \lambda x. e \Leftarrow v_1 \rightarrow \rho$, and by $\text{SB\_DeepSkol}$, we know $\Gamma \vdash_{sb} \lambda x. e \Leftarrow v_1 \rightarrow v_2$.

**Case B\_DLET**

$$\frac{\Gamma \vdash_{sb} e_1 \Rightarrow \sigma_1 \quad \Gamma, x:\sigma_1 \vdash_{sb} e_2 \Leftarrow v}{\Gamma \vdash_{sb} \text{let } x = e_1 \text{ in } e_2 \Leftarrow v} \text{ B\_DLET}$$

By induction we have $\Gamma, x:\sigma_1 \vdash_{sb} e_2 \Leftarrow v$. Also by induction, there is some $\sigma' \leq_b \sigma_1$, such that $\Gamma \vdash_{sb} e \Rightarrow \sigma'$. By context generalization, we know that $\Gamma, x:\sigma'_1 \vdash_{sb} e_2 \Leftarrow v$. So we can use $\text{SB\_DLET}$ to conclude.
Case B_INFER

\[\Gamma \vdash_b e \Rightarrow \sigma_1 \quad \sigma_1 \leq_{dsk} \nu_2 \quad \text{B_INFER}\]

In this case, we know that \(\sigma_1 \leq_{dsk} \nu_2\). We would like to show that \(\Gamma \vdash_{sb} e \Leftarrow \nu_2\). Suppose \(\text{prenex}(\nu_2) = \forall \pi. \rho\), by rule SB_DeepSkol, it suffices to show \(\Gamma \vdash_{sb} e \Leftarrow \rho\).

Suppose \(\sigma_1\) is \(\forall\{\bar{b}\}. \nu_1\). By inversion of \(\sigma_1 \leq_{dsk} \nu_2\), we know that \(\nu_1[\tau/\bar{b}] \leq_{dsk} \rho\).

By induction, there is some \(\sigma' \leq_b \sigma_1\), such that \(\Gamma \vdash^\text{gen}_{sb} e \Rightarrow \sigma'\). By inversion of \(\sigma^\text{gen}\), we know that \(\sigma'\) is of the form \(\forall \{\bar{\pi}\}. \nu'\) and that \(\Gamma \vdash_{sb} e \Rightarrow \nu'\).

By inversion of \(\sigma' \leq_b \sigma_1\), we know that \(\nu'[\tau/\bar{\pi}] \leq_b \nu_1[\tau/\bar{b}]\). By the Barendregt convention, the \(\bar{b}\) can appear in the \(\bar{\pi}\) but not in \(\nu'\). So we can rewrite this substitution as \(\nu'[\tau/\bar{b}] / \bar{\pi} / \bar{\tau}\). We can then substitute (as the \(\bar{\pi}\) were generalized) \(\Gamma \vdash_{sb} e \Rightarrow \nu'[\tau/\bar{b}] / \bar{\pi} / \bar{\tau}\).

By transitivity (Lemma 37), we know that \(\nu'[\tau/\bar{b}] / \bar{\pi} / \bar{\tau} \leq_{dsk} \rho\). So we can conclude \(\Gamma \vdash_{sb} e \Leftarrow \rho\) using SB_INFER.

Case B_SKOL

\[\Gamma \vdash_{b} e \Leftarrow \nu \quad a \notin ftv(\Gamma) \quad \text{B_SKOL}\]

By induction, we have \(\Gamma \vdash_{sb} e \Leftarrow \nu\). We would like to conclude \(\Gamma \vdash_{sb} e \Leftarrow \forall a. \nu\). By inversion, we have \(\Gamma \vdash_{sb} e \Leftarrow \rho\) where \(\text{prenex}(\nu) = \forall \pi. \rho\). However, this means that \(\text{prenex}(\forall a. \nu) = \forall a, \pi. \rho\). 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 [27].
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 [10] (see below). However, it also shows that the deep-skolemization relation is not an essential component of our type system. Instead of $\leq_{dsk}$, it uses the Odersky-Läufer subsumption relation shown in Figure 13.

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 14. The syntax-directed version of the system is in Figure 15. It differs from System SB in that it again uses the $\leq_{ol}$ 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

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

then the expression

$$y \left( \lambda x. (x : a) \right)$$

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,

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

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

\[
\begin{align*}
\tau \leq_{ol} \tau & \quad \text{OL_B_ARFL} \\
\frac{\nu_1 \leq_{ol} \nu_2}{\nu_3 \leq_{ol} \nu_4 \to \nu_1 \to \nu_2 \leq_{ol} \nu_3 \to \nu_4} & \quad \text{OL_B_AFUN} \\
\phi \llbracket \tau/\pi \rrbracket \leq_{ol} \phi & \quad \phi \notin ftv(\forall \pi. \phi_1) \quad \text{OL_B_AIINSTS}
\end{align*}
\]

Fig. 13. Subsumption in the Odersky-Läufer type system
Synthesis rules for System B

\[ \frac{x:\sigma \in \Gamma}{\Gamma \vdash x \Rightarrow \sigma} \quad \text{OL}_B\_\text{VAR} \]

\[ \frac{\Gamma, x:\tau \vdash e \Rightarrow v}{\Gamma \vdash \lambda x. e \Rightarrow \tau \rightarrow v} \quad \text{OL}_B\_\text{ABS} \]

\[ \frac{\Gamma \vdash e_1 \Rightarrow v_1 \rightarrow v_2 \quad \Gamma \vdash e_2 \Rightarrow v_1}{\Gamma \vdash e_1 \, e_2 \Rightarrow v_2} \quad \text{OL}_B\_\text{APP} \]

\[ \frac{\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \Gamma, x:\sigma_1 \vdash e_2 \Rightarrow \sigma}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow \sigma} \quad \text{OL}_B\_\text{LET} \]

\[ \frac{\Gamma \vdash n \Rightarrow \text{Int}}{\Gamma \vdash e \Rightarrow \text{Int}} \quad \text{OL}_B\_\text{INT} \]

\[ \frac{\Gamma \vdash \tau \quad \Gamma \vdash e \Rightarrow \forall a. v}{\Gamma \vdash e @\tau \Rightarrow v[a/\tau]} \quad \text{OL}_B\_\text{TAPP} \]

\[ \frac{\Gamma \vdash v \quad \Gamma \vdash e \Rightarrow v}{\Gamma \vdash e : v \Rightarrow v} \quad \text{OL}_B\_\text{ANNOT} \]

\[ \frac{\Gamma \vdash e \Rightarrow \sigma \quad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e \Rightarrow \forall a. \sigma} \quad \text{OL}_B\_\text{GEN} \]

\[ \frac{\Gamma \vdash e \Rightarrow \sigma_1 \quad \sigma_1 \leq_b \sigma_2}{\Gamma \vdash e \Rightarrow \sigma_2} \quad \text{OL}_B\_\text{SUB} \]

\[ \frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \Rightarrow v} \quad \text{OL}_B\_\text{DLET} \]

\[ \frac{\Gamma \vdash e \Rightarrow v_1 \quad a \notin \text{ftv}(\Gamma)}{\Gamma \vdash e \Rightarrow \forall a. v} \quad \text{OL}_B\_\text{SKOL} \]

\[ \frac{\Gamma \vdash e \Rightarrow v_1 \quad v_1 \leq \omega v_2}{\Gamma \vdash e \Rightarrow v_2} \quad \text{OL}_B\_\text{INFER} \]
Γ, x: τ ⊬ e ⇒ v \rightarrow \text{OL_SB_App}

Γ ⊬ \tau \Rightarrow \text{OL_SB_TApp}

Γ ⊬ v \Rightarrow \text{OL_SB_Annote}

Γ ⊬ \phi \Rightarrow \phi \Rightarrow \text{OL_SB_Phi}

Γ ⊬ e \Rightarrow \sigma \Rightarrow \text{OL_SB_Gen}

Γ ⊬ e \Rightarrow \phi \Rightarrow \text{OL_SB_Skol}

Γ ⊬ e \Rightarrow v \Rightarrow \text{OL_SB_Let}

Γ ⊬ e \Rightarrow \phi \Rightarrow \text{OL_SB_InstS}

Γ ⊬ e \Rightarrow \phi \Rightarrow \text{OL_SB_DAbs}

Γ ⊬ e \Rightarrow \phi \Rightarrow \text{OL_SB_DLet}

Γ ⊬ e \Rightarrow \phi \Rightarrow \text{OL_SB_Infer}

Fig. 15. Syntax-directed specification of System OL-SB
H.1 Properties of OL subsumption

Lemma 43 (Substitution). If \( v_1 \leq_{ol} v_2 \) then \( S(v_1) \leq_{ol} S(v_2) \).

*Proof.* Vytiniotis et al., Lemma 2.1 ([27])

Lemma 44 (Reflexivity for \( \leq_{ol} \)). For all \( v \), \( v \leq_{ol} v \).

*Proof.* Vytiniotis et al., Lemma 2.2 ([27])

Lemma 45 (Transitivity for \( \leq_{ol} \)). If \( v_1 \leq_{ol} v_2 \) and \( v_2 \leq_{ol} v_3 \), then \( v_1 \leq_{ol} v_3 \).

*Proof.* Vytiniotis et al., Lemma 2.3 ([27])

Lemma 46 (Single skol admissible). If \( v_1 \leq_{ol} v_2 \) then \( v_1 \leq_{ol} \forall c. v_2 \) (when \( c \) is not free in \( v_1 \)).

*Proof.* Proof is by induction on \( v_1 \leq_{ol} v_2 \).

Lemma 47 (Monotypes are instantiations). If \( v \leq_{ol} \tau \) then \( v \leq_b \tau \).

*Proof.* Proof is by induction on \( v \leq_{ol} \tau \). In each case the result holds directly by induction.

Lemma 48 (Subsumption contains instantiation). If \( v_1 \leq_b v_2 \) then \( v_1 \leq_{ol} v_2 \).

*Proof.* Proof is by induction.

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

\[
\frac{\phi_1[\tau/b] \leq_{ol} \phi_2}{\forall \pi, \phi_1 \leq_b \forall \pi, \phi_2} \quad \text{B_INSTS}
\]

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

Lemma 49 (Transitivity of Higher-Rank subsumption I). If \( v_1 \leq_v v_2 \) and \( v_2 \leq_{ol} v_3 \), then \( v_1 \leq_{ol} v_3 \).

*Proof.* Follows from Lemmas 45 and 48

Lemma 50 (Transitivity of Higher-Rank subsumption II). If \( v_1 \leq_{ol} v_2 \) and \( v_2 \leq_b v_3 \), then \( v_1 \leq_{ol} v_3 \).

*Proof.* Follows from Lemmas 45 and 48
H.2 Substitution

**Lemma 51 (Substitution for System B).** Assume that the domain of $S$ is disjoint from the variables of $\Gamma$ or the free type variables of $e$.

1. If $\Gamma \vdash_b e \Rightarrow \sigma$ then $\Gamma \vdash_b e \Rightarrow S(\sigma)$
2. If $\Gamma \vdash_b e \Leftarrow v$ then $\Gamma \vdash_b e \Leftarrow S(v)$

**Lemma 52 (Substitution for System SB).** Assume that the domain of $S$ is disjoint from the variables of $\Gamma$ or the free type variables of $e$.

1. If $\Gamma \vdash_{sb} e \Rightarrow \phi$ then $\Gamma \vdash_{sb} e \Rightarrow S(\phi)$
2. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_{sb} e \Rightarrow S(v)$
3. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then $\Gamma \vdash_{sb} e \Rightarrow S(\sigma)$
4. If $\Gamma \vdash_{sb} e \Leftarrow v$ then $\Gamma \vdash_{sb} e \Leftarrow S(v)$
5. If $\Gamma \vdash_{sb} e \Leftarrow \phi$ then $\Gamma \vdash_{sb} e \Leftarrow S(\phi)$

H.3 Other properties

**Lemma 53 (Monotypes are uninformative).** If $\Gamma \vdash_b e \Leftarrow \tau$ then $\Gamma \vdash_b e \Rightarrow \tau$.

*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 $\tau$ is an instantiation of $\sigma$ (lemma 47).

H.4 Soundness of System SB

**Lemma 54 (Soundness of System SB).**

1. If $\Gamma \vdash_{sb} e \Rightarrow \phi$ then $\Gamma \vdash_{sb} e \Rightarrow \phi$.
2. If $\Gamma \vdash_{sb} e \Rightarrow v$ then $\Gamma \vdash_{sb} e \Rightarrow v$.
3. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then $\Gamma \vdash_{sb} e \Rightarrow \sigma$.
4. If $\Gamma \vdash_{sb} e \Leftarrow v$ then $\Gamma \vdash_{sb} e \Leftarrow v$.
5. If $\Gamma \vdash_{sb} e \Leftarrow \phi$ then $\Gamma \vdash_{sb} e \Leftarrow \phi$.

*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:**

\[
\begin{align*}
\Gamma \vdash_{sb} e \Rightarrow v_1 \quad v_1 \leq_{st} \phi_2 \\
\text{no other rule matches} \quad \Gamma \vdash_{sb} e \Leftarrow \phi_2 \\
\hline
\text{OL_SB_INFER}
\end{align*}
\]

By induction, we know that $\Gamma \vdash_b e \Rightarrow v_1$. We can show that $\Gamma \vdash_b e \Leftarrow v_1$ by B_INFER.

**Case SB_SKOL**

\[
\begin{align*}
\Gamma, \pi \vdash_{sb} e \Leftarrow \phi \quad a \notin \text{fv}(\Gamma) \\
\hline
\text{OL_SB_SKOL}
\end{align*}
\]

By induction, we have $\Gamma \vdash_b e \Leftarrow \rho$. We can immediately use rule B_SKOL to conclude.
H.5 Completeness of SB

Generalizing and specializing syntax-directed derivations

Lemma 55 (Context Generalization). Suppose $\Gamma' \leq_b \Gamma$

1. If $\Gamma \vdash_{sb} e \Rightarrow \phi$ then there exists $\phi' \leq_b \phi$ such that $\Gamma' \vdash_{sb} e \Rightarrow \phi'$.
2. If $\Gamma \vdash_{sb} e \Rightarrow \nu$ then there exists $\nu' \leq_b \nu$ such that $\Gamma' \vdash_{sb} e \Rightarrow \nu'$.
3. If $\Gamma \vdash_{sb} e \Rightarrow \sigma$ then there exists $\sigma' \leq_b \sigma$ such that $\Gamma' \vdash_{gen_{sb}} e \Rightarrow \sigma'$.
4. If $\Gamma \vdash_{sb} e \Leftarrow \nu$ and $\nu \leq_b \nu'$ then $\Gamma' \vdash_{sb} e \Leftarrow \nu'$.
5. If $\Gamma \vdash_{sb} e \Leftarrow \phi$ and $\phi \leq_b \phi'$ then $\Gamma' \vdash_{sb} e \Leftarrow \phi$.

Proof. Proof is by induction on derivations.

Case SB_Abs:

\[
\frac{\Gamma, x: \tau \vdash_{sb} e \Rightarrow \nu}{\Gamma \vdash_{sb} \lambda x.e \Rightarrow \tau \Rightarrow \nu} \quad \text{OL} \_SB\_ABS
\]

By induction, we know that $\Gamma', x: \tau \vdash_{sb} e \Rightarrow \nu'$ for $\nu' \leq_b \nu$. Therefore, $\Gamma' \vdash_{sb} \lambda x.e \Rightarrow \tau \Rightarrow \nu'$ and, by SB_FUN, $\tau \Rightarrow \nu' \leq_b \tau \Rightarrow \nu$.

Case SB_Int:

\[
\frac{\Gamma \vdash_{sb} n \Rightarrow \text{Int}}{\Gamma \vdash_{sb} n \Rightarrow \text{Int}} \quad \text{OL} \_SB\_INT
\]

Trivial.

Case SB_Insts:

\[
\frac{\Gamma \vdash_{sb} e \Rightarrow \forall \beta. \phi}{\neg \text{other rule matches} \quad \text{OL} \_SB\_INSTS}
\]

By induction, we know that $\Gamma \vdash_{sb} e \Rightarrow \nu'$ where $\nu' \leq_b \forall \beta. \phi$. By inversion, we know that $\nu'$ must be of the form $\forall \beta. \phi'$ where $\phi' \leq_b \phi$. By SB_Insts, we can conclude $\Gamma \vdash_{sb} e \Rightarrow (\phi' \leq_b \phi)[\beta/\beta]$. We also need to show that $(\phi' \leq_b \phi)[\beta/\beta] \leq_b \phi[\beta/\beta]$, which follows by substitution (Lemma 24).

Case SB_Var:

\[
\frac{x: \varnothing \vdash_{sb} x \Rightarrow \nu}{\Gamma \vdash_{sb} x \Rightarrow \nu} \quad \text{OL} \_SB\_VAR
\]

We know that $x: \sigma \in \Gamma$, where $\sigma \leq_b \forall \varnothing. \nu$. So by inversion, $\sigma$ must be $\forall \{\beta\}. \nu'$ such that $\nu' \leq_b \forall \beta. \phi$. Therefore, by lemma 24, $\nu' \leq_b \nu$. As we know that the $\beta$ are not free in $\nu'$, we can rewrite the left hand side as $\nu'[\beta/\beta]$, and choose those types in the use of SB_Var.

Case SB_App:

\[
\frac{\Gamma \vdash_{sb} e_1 \Rightarrow v_1 \Rightarrow v_2 \quad \Gamma \vdash_{sb} e_2 \Leftarrow v_1}{\Gamma \vdash_{sb} e_1 e_2 \Rightarrow v_2} \quad \text{OL} \_SB\_APP
\]

By induction we have $\Gamma' \vdash_{sb} e_1 \Rightarrow \phi$ such that $\phi \leq_b v_1 \Rightarrow v_2$. By inversion, this means that $\phi$ must be of the form $v'_1 \Rightarrow v'_2$ where $v_1 \leq_b v'_1$ and $v'_2 \leq_b v_2$. By induction, we have $\Gamma' \vdash_{sb} e_2 \Leftarrow v'_1[\beta/\beta]$. So we can conclude by SB_App.
Case SB_Let:

\[
\frac{\Gamma, x: \sigma_1 \mid^*_{sb} e \Rightarrow v_1}{\Gamma \mid^*_{sb} \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2} \quad \text{OL_SB_Let}
\]

By induction \(\Gamma' \mid^*_{sb} e \Rightarrow \sigma'_1\) for \(\sigma'_1 \leq_b \sigma_1\). By induction (again), \(\Gamma', x: \sigma'_1 \mid^*_{sb} e_2 \Rightarrow v'_2\) for \(v'_2 \leq_b v_2\). So we can conclude by SB_Let.

Case SB_TApp:

\[
\frac{\Gamma \mid^* \tau \quad \Gamma \mid^*_{sb} e \Rightarrow \forall a. v}{\Gamma \mid^*_{sb} (e \ : \ v) \Rightarrow v[\tau / a]} \quad \text{OL_SB_TApp}
\]

By induction \(\Gamma' \mid^*_{sb} e \Rightarrow v'\) where \(v' \leq_b \forall a. v\). By inversion, \(v'\) is of the form \(\forall a. v_1\) where \(v_1 \leq_b v\). By substitution, \(v_1[\tau / a] \leq_b v[\tau / a]\).

Case SB_Annot:

\[
\frac{\Gamma \mid^* \psi \quad \Gamma \mid^*_{sb} e \Leftarrow \psi}{\Gamma \mid^*_{sb} (e : \psi) \Rightarrow v} \quad \text{OL_SB_Annot}
\]

Holds directly by induction.

Case SB_Phi:

\[
\frac{\Gamma \mid^* \phi}{\Gamma \mid^*_{sb} e \Rightarrow \phi} \quad \text{no other rule matches} \quad \text{OL_SB_Phi}
\]

Holds directly by induction.

Case SB_Gen:

\[
\pi = \text{ftv}(v) \setminus \text{ftv}(\Gamma) \quad \Gamma \mid^*_{sb} e \Rightarrow v \quad \text{OL_SB_Gen}
\]

By induction, we have \(\Gamma' \mid^*_{sb} e \Rightarrow v'\) for \(v' \leq_b v\). Let \(\bar{b} = \text{ftv}(v') \setminus \text{vars}(\Gamma)\). We want to prove that \(\forall \{\bar{b}\}. v' \leq_b \forall \pi. v\). This holds by definition.

Case SB_DAbs:

\[
\frac{\Gamma, x: \sigma_1 \mid^*_{sb} e \Leftarrow v_1 \quad \Gamma \mid^* \lambda x. e \Leftarrow v_2}{\Gamma \mid^*_{sb} \text{let } x = e_1 \text{ in } e_2 \Leftarrow v_2} \quad \text{OL_SB_DAbs}
\]

Let \(\Gamma' \leq_b \Gamma\) and \(v_1 \rightarrow v_2 \leq_b v'\) be arbitrary. By inversion, we know that \(v'\) is \(v'_1 \rightarrow v'_2\) where \(v'_1 \leq_b v_1\) and \(v'_2 \leq_b v_2\). By induction, we know that \(\Gamma', x: \sigma'_1 \mid^*_{sb} e \Leftarrow v'_2\). So we can conclude by SB_DAbs.

Case SB_Infer:

\[
\frac{\Gamma \mid^*_{sb} e \Rightarrow v_1 \quad v_1 \leq_{ol} \phi_2}{\Gamma \mid^*_{sb} e \Leftarrow \phi_2} \quad \text{OL_SB_Infer}
\]

Let \(\Gamma' \leq_b \Gamma\) and \(\phi_2 \leq_b \phi'\) be arbitrary. By induction, we know that \(\Gamma' \mid^*_{sb} e \Rightarrow v'_1\) for \(v'_1 \leq_b v_1\). We want to show that \(v'_1 \leq_{ol} \phi'\). This holds by lemmas 49 and 50.
Case SB\_DLet:

\[\Gamma \vdash_{\text{gen}} e_1 \Rightarrow \sigma_1 \quad \Gamma, x: \sigma_1 \vdash_{\text{sb}} e_2 \Leftarrow \phi_2 \quad \text{OL\_SB\_DLet}\]

\[\Gamma_{sb} \text{let } x = e_1 \text{ in } e_2 \Leftarrow \phi_2\]

Let \(\Gamma' \leq_b \Gamma\) and \(\phi_2 \leq_b \phi'\) be arbitrary. By induction, we have \(\Gamma' \vdash_{\text{gen}} e_1 \Rightarrow \sigma'_1\) for \(\sigma'_1 \leq_b \sigma_1\). By induction (again), \(\Gamma', x: \sigma'_1 \vdash_{\text{sb}} e_2 \Leftarrow \phi'\). So we can conclude by SB\_DLet.

Case SB\_SKOL:

\[\Gamma, \pi \vdash_{\text{sb}} e \Leftarrow \phi \quad a \notin ftv(\Gamma) \quad \text{OL\_SB\_SKOL}\]

\[\Gamma_{sb} \vdash_{\text{sb}} e \Leftarrow \forall \pi. \phi\]

Let \(\Gamma' \leq_b \Gamma\) be arbitrary. Result holds by directly by induction and SB\_SKOL.

Completeness theorem

Lemma 56 (Completeness of System SB).

1. If \(\Gamma \vdash e \Rightarrow \sigma\) then \(\Gamma \vdash_{\text{gen}} e \Rightarrow \sigma'\) where \(\sigma' \leq_b \sigma\).
2. If \(\Gamma \vdash e \Leftarrow \nu\) then \(\Gamma \vdash_{\text{sb}} e \Leftarrow \nu\).

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

\[x: \sigma \in \Gamma \quad \text{OL\_B\_VAR}\]

\[\Gamma_{sb} \vdash_{\text{sb}} x \Rightarrow \sigma \]

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

Case B\_ABS

\[\Gamma, x: \tau \vdash_{\text{sb}} e \Rightarrow \nu \quad \text{OL\_B\_ABS}\]

\[\Gamma_{sb} \vdash_{\text{sb}} \lambda x. e \Rightarrow \tau \Rightarrow \nu \]

The induction hypothesis gives us \(\Gamma, x: \tau \vdash_{\text{gen}} e \Rightarrow \forall \{\pi\}. \nu'\) where \(\forall \{\pi\}. \nu' \leq_b \nu\). Inverting \(\vdash_{\text{gen}}\) gives us \(\Gamma, x: \tau \vdash_{\text{sb}} e \Rightarrow \nu'\) and inverting \(\leq_b\) gives us \(\nu'[\pi/\pi] \leq_b \nu\). We can then substitute and use SB\_ABS to get \(\Gamma_{sb} \vdash_{\text{sb}} \lambda x. e \Rightarrow \tau \Rightarrow \nu'[\pi/\pi]\). Generalizing, we get \(\Gamma_{sb} \vdash_{\text{gen}} \lambda x. e \Rightarrow \forall \{\pi, \pi'\}. \tau \Rightarrow \nu'[\pi/\pi]\) where the new variables \(\pi'\) come from generalizing \(\tau\) and the \(\pi\). We are done because \((\tau \Rightarrow \nu'[\pi/\pi]) \leq_b \tau \Rightarrow \nu\) and so \(\forall \{\pi, \pi'\}. \tau \Rightarrow \nu' \leq_b \tau \Rightarrow \nu\).
Case B _App

\[ \frac{\Gamma \vdash_\beta e_1 \Rightarrow v_1 \rightarrow v_2 \quad \Gamma \vdash_\beta e_2 \leftarrow v_1}{\Gamma \vdash_\beta e_1 \ e_2 \Rightarrow v_2} \]

By induction we have \( \Gamma \vdash_\beta \sigma \Rightarrow e_1 \) for some \( \sigma \leq_\beta v_1 \rightarrow v_2 \).
So by inversion \( \sigma = \forall \{ \pi \}. b. v_1' \rightarrow v_2' \), where \( (v_1' \rightarrow v_2')[\tau / \pi][\tau' / \bar{b}] \leq_\beta v_1 \rightarrow v_2 \) and \( v_1 \leq_\beta v_1'[\tau / \pi][\tau' / \bar{b}] \) and \( v_2'[\tau / \pi][\tau' / \bar{b}] \leq_\beta v_2 \).

Also by inversion, we know that \( \Gamma \vdash_\beta \sigma_1 \leftarrow v_1 \) and \( \sigma_2 \) by Lemma 24.

By substitution, we have \( \Gamma \vdash_\beta \sigma \leftarrow v_1 \) and that the \( \pi \) are not free in \( \Gamma \) or \( e_1 \).

By substitution, we have \( \Gamma \vdash_\beta e_1 \Rightarrow \forall \{ \pi \}. (v_1' \rightarrow v_2')[\tau / \pi] \)

And by SB _Inst , we have \( \Gamma \vdash_\beta e_1 \Rightarrow (v_1' \rightarrow v_2')[\tau / \pi][\tau' / \bar{b}] \).

By induction we have \( \Gamma \vdash_\beta e_2 \leftarrow v_1 \) and by lemma 55 we know that \( \Gamma \vdash_\beta e_2 \leftarrow v_1 \).

So we can conclude \( \Gamma \vdash_\beta e_1 \ e_2 \leftarrow v_1'[\tau / \pi][\tau' / \bar{b}] \).

Case B _Let

\[ \frac{\Gamma \vdash_\beta e_1 \Rightarrow \sigma \quad \Gamma, x : \sigma_1 \vdash_\beta e_2 \Rightarrow \sigma}{\Gamma \vdash_\beta \text{let } x = e_1 \text{ in } e_2 \Rightarrow \sigma} \]

The induction hypothesis gives us \( \Gamma \vdash_\beta e_1 \Rightarrow \sigma' \) with \( \sigma' \leq_\beta \sigma_1 \).

The induction hypothesis also gives us \( \Gamma, x : \sigma_1 \vdash_\beta e_2 \Rightarrow \sigma_2 \) with \( \sigma_2 \leq_\beta \sigma_2 \).

Use Lemma 55 to get \( \Gamma, x : \sigma_1 \vdash_\beta e_2 \Rightarrow \sigma_2 \)

Let \( \sigma_2' = \forall \{ \pi \}. v \) where \( \bar{b} = \text{ftv}(v) \setminus \text{vars}(\Gamma) \).

Inverting \( \vdash_\beta \) gives us \( \Gamma, x : \sigma_1 \vdash_\beta e_2 \Rightarrow v \).

We then use SB _Let to get \( \Gamma \vdash_\beta \text{let } x = e_1 \text{ in } e_2 \Rightarrow v \).

Generalizing gives us \( \Gamma \vdash_\beta \text{let } x = e_1 \text{ in } e_2 \Rightarrow \forall \{ \pi \}. v \).

Transitivity of \( \leq_\beta \) (Lemma 20) gives us \( \forall \{ \pi \}. v \leq_{hmv} \sigma_2 \).

Case B _Int

\[ \frac{\Gamma \vdash_\beta n \Rightarrow \text{Int}}{\Gamma \vdash_\beta \text{Int}} \]

Trivial.

Case B _TApp

\[ \frac{\Gamma \vdash_\beta \tau \quad \Gamma \vdash_\beta e \Rightarrow \forall a. v \quad \Gamma \vdash_\beta e \ @ \tau \Rightarrow v[\tau / a]}{\Gamma \vdash_\beta e \ @ \tau \Rightarrow \forall a. v} \]

The induction hypothesis (after inverting \( \vdash_\beta \)) gives us \( \Gamma \vdash_\beta e \Rightarrow \forall a. v' \),

where \( \bar{b} = \text{ftv}(\forall a. v') \setminus \text{vars}(\Gamma) \) and \( v'[\tau / \bar{b}] \leq_\beta v \).

Applying SB _TApp gives us \( \Gamma \vdash_\beta e \ @ \tau \Rightarrow v'[\tau / a] \), and SB _Gen gives us \( \Gamma \vdash_\beta e \ @ \tau \Rightarrow \forall \{ \pi \}. v'[\tau / a] \).

We want to show that \( \forall \{ \pi \}. v'[\tau / a] \leq_\beta v[\tau / a] \), which follows when there is some \( \tau' \), such that \( v'[\tau / a][\tau / \tau'] \leq_\beta v[\tau / a] \).

This is equivalent to exchanging the substitution, i.e. finding a \( \tau' \) such that \( v'[\tau / \bar{b}][\tau / a] \leq_\beta v[\tau / a] \).

By Substitution (Lemma 24), we have \( v'[\tau / \bar{b}][\tau / a] \leq_\beta v[\tau / a] \).

We also know that the \( \bar{b} \) are a subset of the \( \tau \). So we can choose \( \tau' \) to be \( \tau \) for the \( \bar{b} \), and the remaining \( \tau \) elsewhere, and we are done.
Case B_Annot

\[
\frac{
\Gamma \vdash v \\
\Gamma \vdash_b e \Leftarrow v
}{
\Gamma \vdash_b (e : v) \Rightarrow v}
\]

OL_B_Annot

The induction hypothesis gives us \( \Gamma \vdash_b e \Leftarrow v \), so the result immediately follows.

Case B_Gen

\[
\frac{
\Gamma \vdash_e e \Rightarrow \sigma \\
\Gamma \vdash_b e \Rightarrow \forall \{a\}. \sigma
}{
\Gamma \vdash_b e \Rightarrow \sigma}
\]

OL_B_Gen

The induction hypothesis gives us \( \Gamma \vdash_{\text{gen}} e \Rightarrow \sigma' \) where \( \sigma' \leq_b \sigma \). We know \( \sigma' \leq_b \forall \{a\}. \sigma \). In other words, if \( \sigma' = \forall \{b\}. \nu_1 \) and \( \sigma = \forall \{\tau\}. \nu_2 \), we have some \( \tau \) such that \( \nu_1[\tau/b] = \nu_2 \). By the definition of \( \leq_b \) we can use these same \( \tau \) to show that \( \sigma' \leq_b \forall \{a, \tau\}. \nu_2 \).

Case B_Sub

\[
\frac{
\Gamma \vdash_b e \Rightarrow \sigma_1 \\
\Gamma \vdash_b e \Rightarrow \forall \{a\}. \sigma_2
}{
\Gamma \vdash_b e \Rightarrow \sigma_1 \leq_b \sigma_2}
\]

OL_B_Sub

The induction hypothesis gives us \( \Gamma \vdash_{\text{gen}} e \Rightarrow \sigma' \) where \( \sigma' \leq_b \sigma_1 \). By transitivity of \( \leq_b \), we are done.

Case B_DAbs

\[
\frac{
\Gamma \vdash_b e \Rightarrow \sigma_1 \\
\Gamma \vdash_b e \Rightarrow \forall \{a\}. \sigma_2
}{
\Gamma \vdash_b \lambda x. e \Rightarrow \forall \{a\}. \sigma_2}
\]

OL_B_DAbs

By induction, we have that \( \Gamma, x : \sigma_1 \vdash_{\text{gen}} e \Rightarrow \nu_2 \). Therefore by SB_DAbs, we can conclude \( \Gamma, x : \sigma_1 \vdash_{\text{gen}} \lambda x. e \Rightarrow \nu_1 \Rightarrow \nu_2 \).

Case B_DLet

\[
\frac{
\Gamma \vdash_b e \Rightarrow v_1 \\
\Gamma \vdash_{\text{gen}} e_1 \Rightarrow \sigma_1 \\
\Gamma \vdash_{\text{gen}} e_2 \Rightarrow \nu_2
}{
\Gamma \vdash_b \text{let } x = e_1 \text{ in } e_2 \Rightarrow \nu_2}
\]

OL_B_DLet

By induction we have \( \Gamma, x : \sigma_1 \vdash_{\text{gen}} e \Rightarrow \nu_2 \). Also by induction, there is some \( \sigma' \leq_b \sigma_1 \), such that \( \Gamma \vdash_{\text{gen}} e \Rightarrow \sigma' \). By context generalization, we know that \( \Gamma, x : \sigma'_1 \vdash_{\text{gen}} e_2 \Rightarrow \nu_2 \). So we can use SB_DLet to conclude.

Case B_Skol

\[
\frac{
\Gamma, a \vdash_b e \Leftarrow v \\
\Gamma \vdash_{\text{gen}} e \Rightarrow \forall a. \nu
}{
\Gamma, a \vdash_b e \Leftarrow \forall a. \nu}
\]

OL_B_Skol

By induction, we have \( \Gamma, a \vdash_{\text{gen}} e \Leftarrow v \). We can conclude using SB_Skol. (Actually, first inverting and then potentially applying SB_Skol multiple times).

Case B_Infer

\[
\frac{
\Gamma \vdash_b e \Rightarrow \nu_1 \\
\Gamma \vdash_{\text{gen}} e \Rightarrow \forall a. \nu_2
}{
\Gamma \vdash_b e \Leftarrow \forall a. \nu_2}
\]

OL_B_Infer
By induction, there is some \( \sigma' \leq_b \upsilon_1 \), such that \( \Gamma \vdash_{\text{gen}} \upsilon \Rightarrow \sigma' \). By inversion of \( \vdash_{\text{gen}} \), we know that \( \sigma' \) is of the form \( \forall \{ \overline{a} \} \cdot \upsilon' \) and that \( \Gamma \vdash_{\text{inst}} e \Rightarrow \upsilon' \), where \( \overline{a} \) do not appear in \( e \) and \( \Gamma \).

By inversion of \( \leq_b \), we know that \( \upsilon'[\overline{\tau}/a] \leq_b \upsilon_1 \). By substitution, we know that \( \Gamma \vdash_{\text{inst}} e \Rightarrow \upsilon'[\overline{\tau}/a] \).

By transitivity (Lemma 49) we know that \( \upsilon'[\overline{\tau}/a] \leq_{\text{ol}} \upsilon_2 \). So we can conclude using \( \text{SB}_\text{Infer} \).

H.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 \( \Gamma \vdash_{\text{DK}} \upsilon_1 \leq \upsilon_2 \) is the subtyping relation from the DK paper, Figure 1.

**Lemma 57 (Higher-rank subsumption contains DK subtyping).** If \( \Gamma \vdash_{\text{DK}} \upsilon_1 \leq \upsilon_2 \) then \( \upsilon_1 \leq_{\text{ol}} \upsilon_2 \).

**Proof.** But induction on the DK subtyping judgement.

- **Case Decl\leq\text{Var}** Immediate from DSK\_REFL.
- **Case Decl\leq\text{Unit}** Immediate from DSK\_REFL.
- **Case Decl\leq\text{\rightarrow}** Directly via induction and DSK\_FUN.
- **Case Decl\leq\forall L** By induction and DSK\_INST.
- **Case Decl\leq\forall R** By induction and DSK\_INST.

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

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 58.** If \( \Gamma \vdash_{\text{DK}} \upsilon_1 \circ e \Rightarrow \Rightarrow \upsilon_2 \) then there exists some \( \upsilon'_1 \) such that \( \upsilon_1 \leq_b \upsilon'_1 \to \upsilon_2 \) and \( \Gamma \vdash_{\text{DK}} e \Leftarrow \upsilon'_1 \).

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

Now let \( \Gamma \vdash_{\text{DK}} e \Rightarrow \upsilon, \Gamma \vdash_{\text{DK}} e \Leftarrow \upsilon \) be the judgements shown in Figure 4 of their paper. We can also argue that System B can typecheck the same terms.
Lemma 59.

1. If $\Gamma \vdash_{DK} e \bowtie v$ then $\Gamma \vdash_{B} e \bowtie v$.
2. If $\Gamma \vdash_{DK} e \Rightarrow v$ then $\Gamma \vdash_{B} e \Rightarrow v$.

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 53 to convert a checking judgement for monotypes into a synthesis judgement.