Local Type Argument Synthesis with Bounded Quantification

Benjamin C. Pierce
Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, IN 47405, USA
pierce@cs.indiana.edu

David N. Turner
An Teallach Limited
Technology Transfer Center
King's Buildings
Edinburgh, EH9 3JL, UK
dnt@an-teallach.com

Indiana University
CSCI Technical Report #495

November 12, 1997

Abstract

In a companion paper [PT98], we introduced a *local type inference* method for inferring type arguments to polymorphic functions. We show here how our method can be extended to handle Cardelli and Wegner's Kernel Fun variant of $F_{\leq}$.

1 Introduction

Most statically typed programming languages offer some form of *type inference*, allowing programmers to omit type annotations that can be recovered from context. Such a facility can eliminate a great deal of needless verbosity, making programs easier both to read and to write. Unfortunately, type inference technology has not kept pace with developments in type systems. In particular, the combination of subtyping and parametric polymorphism has been intensively studied for more than a decade in calculi such as System $F_{\leq}$ [CW85, CG92, CMMS94, etc.], but these features have not yet been satisfactorily integrated with practical type inference methods. Part of the reason for this gap is that most work on type inference for this class of languages has concentrated on the difficult problem of developing *complete* methods, which are guaranteed to infer types, whenever possible, for entirely unannotated programs. In a companion paper [PT98], we proposed a simple alternative, refining the idea of *partial* type inference with the additional simplifying principle that missing annotations should be recovered using only types propagated *locally*, from adjacent nodes in the syntax tree.

In [PT98], we presented our type inference scheme for a calculus incorporating both impredicative polymorphism and subtyping. For simplicity, we did not treat *bounded quantification*, where subtyping and quantification interact via upper bounds on type variables. This paper rectifies that shortcoming, showing how our type inference method can be extended to handle Cardelli and Wegner's Kernel Fun variant of $F_{\leq}$.

There is one caveat: we make some restrictions on the kinds of polymorphic functions we automatically infer type arguments for. In particular, we have so far been unable to deal with inter-dependent bounds: we do not know of a complete algorithm which can synthesize, for example, the type arguments for a function of type `All(X<:Top,Y<:X)S→T`. Rather than introduce a potentially unimplementable rule in the specification of type inference, we explicitly disallow this case in our specification: the user must always write explicit type arguments on applications of such functions.

We concentrate here on the technical details of how to extend our local type inference scheme to allow bounded quantification. We refer the reader to [PT98] for a fuller explanation of the basic local type inference scheme. In the next section, we define (a mild extension of) Cardelli and Wegner's Kernel Fun. Section 3 then shows how we can specify and implement a local type argument inference scheme for this calculus.

2 Kernel Fun

For our explicitly-typed internal language—the target for the type inference methods described in Section 3—we extend Cardelli and Wegner's Kernel Fun calculus [CW85] of subtyping and impredicative polymorphism. We only give definitions here; the meta-theory of the system is developed in detail in a companion paper [Pie97].

2.1 Syntax

We extend the original Kernel Fun system [CW85] in a few significant ways.

Firstly, we add a minimal type `Bot`. Our type inference algorithm keeps track of various type constraints by calculating the least upper bound and greatest lower bound of pairs of types. The `Bot` type plays a crucial role in these calculations, since without it we could not guarantee that least upper-bounds and greatest lower-bounds always exist. The properties of Kernel Fun with `Bot` are similar to those of pure Kernel Fun, but there are a number of significant differences in details. The properties of the internal language are developed in detail in [Pie97].

Secondly, we extend abstraction and application so that several arguments (including both types and terms) may be passed at the same time. In other words, we favor a "fully uncurried" style of function definition and application (though currying is, of course, still available). This bias will play an important role in our scheme for inferring type arguments in Section 3.

The syntax of types, terms, and typing contexts in the internal language is as follows:

$\mathtt{T}$	$::=$	$\mathtt{X}$	type variable
		$\mathtt{Top}$	maximal type
		$\mathtt{Bot}$	minimal type
		$\mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{T}})\overline{\mathtt{T}}\to\mathtt{T}$	function type
$\mathtt{e}$	$::=$	$\mathtt{x}$	variable
		$\mathtt{fun}[\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{T}}](\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}})\mathtt{e}$	abstraction
		$\mathtt{e}[\overline{\mathtt{T}}](\overline{\mathtt{e}})$	application
,	$::=$	$\bullet$	empty context
		$, ,\mathtt{x:T}$	variable binding
		$, ,\mathtt{X<:T}$	type variable binding

We use the meta-variables `A`, `B`, `R`, `S`, `T` and `U` to range over types; `e` and `f` range over terms. We use the notation $\overline{\mathtt{X}}$ to denote the sequence $\mathtt{X}_1,\ldots,\mathtt{X}_n$, and similarly $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$ to denote $\mathtt{x}_1\mathtt{:T}_1,\ldots,\mathtt{x}_n\mathtt{:T}_n$. We write , (`X`) for the bound of `X` in , and , (`x`) for the type of `x` in , .

We write $\overline{\mathtt{S}}\to\mathtt{T}$ as an abbreviation for the monomorphic function type $\mathtt{All()}\overline{\mathtt{S}}\to\mathtt{T}$. Similarly, we write $\mathtt{fun}(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}})\mathtt{e}$ as an abbreviation for the monomorphic function $\mathtt{fun[]}(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}})\mathtt{e}$.

Types, terms, contexts, and judgments that differ only in the names of bound variables are regarded as identical. Binders in contexts are assumed to have distinct names; when a new binding is added to a context, we assume that it has been renamed so as to maintain this invariant. The rules for scoping of bound variables are as usual (in $\mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{S}})\overline{\mathtt{T}}\to\mathtt{U}$, the scope of each $\mathtt{X}_i$ is everything to its right: the bounds $\mathtt{S}_{i+1}$ through $\mathtt{S}_n$, all of $\overline{\mathtt{T}}$, and `U`). $FV(\mathtt{T})$, the set of type variables free in `T`, is defined in the usual way.

2.2 Subtyping

The subtyping relation is the evident extension of Kernel Fun's subtyping relation with the `Bot` type. We write $, \vdash \overline{\mathtt{S}}$ <: $\overline{\mathtt{T}}$ to mean "$|\overline{\mathtt{S}}| = |\overline{\mathtt{T}}|$ and $, \vdash \mathtt{S}_i$ <: $\mathtt{T}_i$ for all $1 \le i \le |\mathtt{S}|$."

$$, \vdash \mathtt{T} \mathrel{<:} \mathtt{Top} \qquad \text{(S-Top)}$$

$$, \vdash \mathtt{Bot} \mathrel{<:} \mathtt{T} \qquad \text{(S-Bot)}$$

$$, \vdash \mathtt{X} \mathrel{<:} \mathtt{X} \qquad \text{(S-Refl)}$$

$$\frac{, \vdash , (\mathtt{X}) \mathrel{<:} \mathtt{T}}{, \vdash \mathtt{X} \mathrel{<:} \mathtt{T}} \qquad \text{(S-Var)}$$

$$\frac{, , \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}} \vdash \overline{\mathtt{T}} \mathrel{<:} \overline{\mathtt{R}} \qquad , , \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}} \vdash \mathtt{S} \mathrel{<:} \mathtt{U}}{, \vdash \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}})\overline{\mathtt{R}}\rightarrow\mathtt{S} \mathrel{<:} \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}})\overline{\mathtt{T}}\rightarrow\mathtt{U}} \qquad \text{(S-Fun)}$$

Two points are worth noting. First, we use the original "Kernel Fun" rule for comparing quantifiers [CW85], in which the upper bounds $\overline{\mathtt{B}}$ in the subtyping rule for polymorphic functions are required to be identical, rather than the more powerful but less tractable variant of Curien and Ghelli [CG92, CMMS94].[1] The principal reason for this restriction is that it allows us to define meets and joins of all pairs of types, which may fail to exist in "Full $F_{\le}$" [Ghe90]. Second, for simplicity, we use an algorithmic presentation of subtyping, in which the rules of transitivity and general reflexivity are omitted and recovered as properties of the definition (cf. [Pie97, Section 3.1]).

It is also important to note that some of the usual properties of presentations of Kernel Fun without `Bot` do not hold here. For instance, $, \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ and $, \vdash \mathtt{T} \mathrel{<:} \mathtt{S}$ do not imply $\mathtt{S} = \mathtt{T}$ (consider, for example, $\mathtt{X}\mathtt{<:}\mathtt{Bot} \vdash \mathtt{X} \mathrel{<:} \mathtt{Bot}$ and $\mathtt{X}\mathtt{<:}\mathtt{Bot} \vdash \mathtt{Bot} \mathrel{<:} \mathtt{X}$). This fact is the result of an unfortunate interaction between bounded quantification and `Bot`, and it substantially complicates the proofs of the properties in the remainder of this section as well as the development in Section 3.

We write $, \vdash \mathtt{S} \Uparrow \mathtt{T}$ for the *least non-variable supertype* of `S`, defined by repeated promotion of variables:

$$\frac{\mathtt{S} \text{ is not a variable}}{, \vdash \mathtt{S} \Uparrow \mathtt{S}}$$

$$\frac{, \vdash , (\mathtt{X}) \Uparrow \mathtt{T}}{, \vdash \mathtt{X} \Uparrow \mathtt{T}}$$

We write $, \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ for "`M` is the meet of `S` and `T` in context , " and $, \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$ for "`J` is the join of `S` and `T` in , ." The definitions of these relations can be found in [Pie97], Section 3.3.

2.3 Typing Rules

The typing relation $, \vdash \mathtt{e} \in \mathtt{T}$ is essentially the standard one, except that, as in the definition of subtyping, we use an algorithmic presentation, omitting the usual rule of subsumption ("if $\mathtt{e} \in \mathtt{S}$ and $\mathtt{S} \mathrel{<:} \mathtt{T}$, then $\mathtt{e} \in \mathtt{T}$"); instead, the rules below calculate for each typable term a single *manifest type*, corresponding to its minimal type in the system with subsumption. For subtyping, the choice of algorithmic presentation was made for the sake of simplicity. Here, it is actually crucial: our type inference methods depend on the fact that a typable term has a unique type, and that this type can easily be predicted by the programmer. (Note that this stylistic choice does not change the set of typable terms.)

In the rules, we write $, \vdash \mathtt{e} \Uparrow \mathtt{T}$ to abbreviate $, \vdash \mathtt{e} \in \mathtt{S}$ and $, \vdash \mathtt{S} \Uparrow \mathtt{T}$; similarly, $, \vdash \mathtt{e} \in \mathtt{S} \mathrel{<:} \mathtt{T}$ abbreviates $, \vdash \mathtt{e} \in \mathtt{S}$ and $, \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.

[1] A variant on the rule used here would require that the upper bounds be *equivalent*—i.e., each a subtype of the other. Choosing this variant appears to make some of the following development simpler and other parts more complex, sometimes substantially so. It is not clear to us which is globally better.

The typing rule for variables is standard.

$$, \vdash \texttt{x} \in , (\texttt{x}) \qquad \text{(T-VAR)}$$

The rule for (multi-)abstractions combines the usual rules for term and type abstractions.

$$\frac{, , \overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}, \overline{\texttt{x}}\texttt{:}\overline{\texttt{S}} \vdash \texttt{e} \in \texttt{T}}{, \vdash \texttt{fun[}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}\texttt{](}\overline{\texttt{x}}\texttt{:}\overline{\texttt{S}}\texttt{)e} \in \texttt{All(}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}\texttt{)}\overline{\texttt{S}}\texttt{→T}} \qquad \text{(T-ABS)}$$

Similarly, the rule for (multi-)applications combines the usual application and polymorphic application rules of Kernel Fun. We calculate the type of the function `f` and promote it (if it is a variable), to an explicit function type $\texttt{All(}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}\texttt{)}\overline{\texttt{S}}\texttt{→R}$. We then check that the provided type arguments are subtypes of the upper bounds $\overline{\texttt{B}}$ and that the provided term arguments have the expected types. If all these constraints are satisfied, then the result type of the application is found by substituting the actual type arguments into `R`.

$$\frac{\begin{array}{c}, \vdash \texttt{f} \Uparrow \texttt{All(}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}\texttt{)}\overline{\texttt{S}}\texttt{→R} \\ , \vdash \texttt{T}_i \texttt{ <: } [\texttt{T}_1/\texttt{X}_1 \dots \texttt{T}_{i-1}/\texttt{X}_{i-1}]\texttt{B}_i \qquad , \vdash \overline{\texttt{e}} \in \overline{\texttt{U}} \texttt{ <: } [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\end{array}}{, \vdash \texttt{f[}\overline{\texttt{T}}\texttt{](}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{R}} \qquad \text{(T-APP)}$$

To finish the definition of the typing relation, another rule is required. To see why, note that $\texttt{Bot <: All(}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{B}}\texttt{)}\overline{\texttt{S}}\texttt{→T}$ for any $\overline{\texttt{X}}$, $\overline{\texttt{B}}$, $\overline{\texttt{S}}$, and `T`. This means that any expression of type `Bot` should be applicable to any set of well-formed type and expression arguments (if we did not allow for this behavior, we would lose the type soundness property):

$$\frac{, \vdash \texttt{f} \Uparrow \texttt{Bot} \qquad , \vdash \overline{\texttt{e}} \in \overline{\texttt{S}}}{, \vdash \texttt{f[}\overline{\texttt{T}}\texttt{](}\overline{\texttt{e}}\texttt{)} \in \texttt{Bot}} \qquad \text{(T-BOT)}$$

Note that this rule gives the expression $\texttt{f[}\overline{\texttt{T}}\texttt{](}\overline{\texttt{e}}\texttt{)}$ result type `Bot`, the most informative result type for the expression.

2.3.1 Theorem [Uniqueness of manifest types]: Each typable term has a unique type, up to equivalence: if $, \vdash \texttt{e} \in \texttt{S}$ and $, \vdash \texttt{e} \in \texttt{T}$, then $, \vdash \texttt{S <: T}$ and $, \vdash \texttt{T <: S}$. (Indeed, given the definitions we have seen so far, it is even the case that $\texttt{S} = \texttt{T}$; we state the proposition in this weaker version because later we will want to guarantee uniqueness of inferred types only up to equivalence in the subtype relation.)

The definitions of operational and denotational semantics for the above calculus are standard, as are proofs of properties such as subject reduction and absence of runtime errors. Evaluation order may be chosen either call-by-name or call-by-value; function spaces may be interpreted as either total or partial. The only slightly unusual case is the type `Bot`, which can be interpreted as an empty type (in a total-function semantics) or a type containing only divergent terms (in a partial function semantics).

3 Local Type Argument Synthesis

Our measurements of ML programs in [PT98] showed that type arguments to polymorphic functions are inferred by the ML typechecker on at least one line in every three, in typical programs. Moreover, explicit type arguments rarely have any useful documentation value. As an example, consider the polymorphic identity function `id` with type `All(X<:Top)X→X`. Our goal is to allow the programmer to apply the `id` function without explicitly supplying any type arguments: `id(3)` rather than `id[Int](3)`.

The main problem we have to address is the fact that, in general, there may be a number of different type arguments that we can pick for a particular function application. For example, both `id[Int](x)` and `id[Real](x)` are valid completions of the term `id(x)`, where $\texttt{x} \in \texttt{Int}$ and `Int` is a subtype of `Real`. Fortunately, there is usually a good way to choose between all the alternatives: we pick the type arguments that yield the best (smallest) type for the result. In the case of `id(x)`, we choose `id[Int](x)` since this has result type `Int`, which is more informative type than the result type `Real` of `id[Real](x)`.

Sadly, there are cases where there is no best result type. Suppose, for example, that `f` has type `All(X<:Top)()`$\to$`(X`$\to$`X)` (a function which takes a single type argument `X` and returns a function from `X` to `X`). Two possible completions of the term `f()` are `f[Int]()` and `f[Real]()`, which have result types `Int`$\to$`Int` and `Real`$\to$`Real`. These two result types are incomparable in the subtyping relation, so there is no "best" result type available. In this case type argument synthesis will fail, since it is not possible to locally determine the missing type arguments for `f` (in [PT98] we show how propagating additional contextual information sometimes allows us to avoid this situation).

3.1 Type Inference (Specification)

We define a three-place *type inference* relation:

$$, \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}'$$

Intuitively, this relation can be read "In context , , type annotations can be added to the term `e` to yield the fully-typed term $\mathtt{e}'$, which has type `T`."

The specification of the type inference relation is quite simple. For each typing rule with conclusion $, \vdash \mathtt{e} \in \mathtt{T}$ in the explicitly-typed language, the type inference relation contains an analogous rule with conclusion $, \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}'$, where $\mathtt{e}'$ is derived in the obvious way from the fully typed subexpressions yielded by subderivations. To these rules is added one additional rule, handling the case where type arguments are omitted:

$$\frac{\begin{array}{c} , \vdash \mathtt{f} \Uparrow \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{S}})\overline{\mathtt{T}}\to\mathtt{R} \Rightarrow \mathtt{f}' \qquad , \vdash \overline{\mathtt{e}} \in \overline{\mathtt{U}} \Rightarrow \overline{\mathtt{e}}' \\ |\overline{\mathtt{X}}| > 0 \qquad \overline{\mathtt{X}} \cap FV(\overline{\mathtt{S}}) = \emptyset \qquad , \vdash \overline{\mathtt{A}} \mathrel{\mathtt{<:}} \overline{\mathtt{S}} \qquad , \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{A}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \\ \forall \overline{\mathtt{B}}.\quad (, \vdash \overline{\mathtt{B}} \mathrel{\mathtt{<:}} \overline{\mathtt{S}} \ \text{ and } \ , \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{B}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \ \text{ imply } \ , \vdash [\overline{\mathtt{A}}/\overline{\mathtt{X}}]\mathtt{R} \mathrel{\mathtt{<:}} [\overline{\mathtt{B}}/\overline{\mathtt{X}}]\mathtt{R}) \end{array}}{, \vdash \mathtt{f}(\overline{\mathtt{e}}) \in [\overline{\mathtt{A}}/\overline{\mathtt{X}}]\mathtt{R} \Rightarrow \mathtt{f}'[\overline{\mathtt{A}}](\overline{\mathtt{e}}')} \qquad \text{(App-InfSpec)}$$

The condition $|\overline{\mathtt{X}}| > 0$ says that type argument synthesis is only required in the case where the function `f` is polymorphic but there are no explicit type arguments. (For simplicity, we don't synthesize type arguments in the case where an application node provides some, but not all, of its required type arguments explicitly. This would be easy to do, but does not seem very useful in practice.) The condition $\overline{\mathtt{X}} \cap FV(\overline{\mathtt{S}}) = \emptyset$ explicitly disallows type argument synthesis in the case where the bounds $\overline{\mathtt{S}}$ are inter-dependent (since, at this time, we do not know of a complete solution to this problem).

The type arguments $\overline{\mathtt{A}}$ that we pick as the result of our synthesis rule must satisfy a number of conditions. Firstly, they must must be legal type arguments for `f`. (The condition $, \vdash \overline{\mathtt{A}} \mathrel{\mathtt{<:}} \overline{\mathtt{S}}$ ensures that the arguments are subtypes of the required bounds $\overline{\mathtt{S}}$, while the condition $, \vdash \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{A}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$ ensures that the types of the argument expressions match the types of the function parameters.) Secondly, the final line of the rule asserts that the arguments $\overline{\mathtt{A}}$ must be chosen in such a way that any other choice of arguments $\overline{\mathtt{B}}$ satisfying the above conditions will yield a less informative result type, i.e., a supertype of $[\overline{\mathtt{A}}/\overline{\mathtt{X}}]\mathtt{R}$.

To state the formal properties of this technique, we need to relate terms in the internal language to terms in the external language. We say that a term `e` is a *partial erasure* of $\mathtt{e}'$ if `e` can be obtained from $\mathtt{e}'$ by erasing some type annotations (i.e., deleting type arguments from one or more applications).

3.1.1 Theorem [Soundness]:

If $, \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}'$, then `e` is a partial erasure of $\mathtt{e}'$ and $, \vdash \mathtt{e}' \in \mathtt{T}$.

Proof: Straightforward from the definition. ∎

Since we are dealing with a partial type inference technique, we cannot expect a completeness property at this point. However, we can show that the type inference relation is "*locally* complete" in the sense that its specification guarantees that it will find the best values for missing type arguments in a single application, whenever these exist.

3.1.2 Theorem [Partial Completeness]: If $, \vdash \mathtt{e} \in \mathtt{T}$ (i.e., e is fully typed), then $, \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}$.

Proof: Straightforward since, in inferring types for a fully typed program, the local type argument synthesis rule can never be used. ∎

It should be emphasized that the App-InfSpec rule, together with the other typing rules from the explicitly-typed language, constitutes a complete specification of the type inference relation: it is all that a programmer needs to understand in order to use the language. Only the compiler writer needs to go further into the development in the rest of the section, whose job is to show how the rule we have given can be implemented.

3.2 Variable Elimination

In the constraint-generation algorithm that we present in the next section, it will sometimes be necessary to eliminate all occurrences of a certain set of variables from a given type by promoting or demoting the type until we reach a type in which these variables do not occur. (This property is another crucial reason for choosing the "Kernel Fun" variant of $F_{\leq}$ rather than the "full $F_{\leq}$" variant where two polymorphic function types with different upper bounds for their type components are allowed to stand in the subtype relation under appropriate conditions; in the latter system, it can be shown that variables cannot always be eliminated in a most general way [GP97].)

Formally, we write $, \vdash \mathtt{S} \Uparrow^V \mathtt{T}$ for the relation "T is the least supertype of S such that $FV(\mathtt{T}) \cap V = \emptyset$" and $, \vdash \mathtt{S} \Downarrow_V \mathtt{T}$ for the dual relation "T is the greatest subtype of S such that $FV(\mathtt{T}) \cap V = \emptyset$." The variable-elimination-by-promotion relation can be computed as follows:

$$, \vdash \mathtt{Top} \Uparrow^V \mathtt{Top} \qquad \text{(VU-Top)}$$

$$, \vdash \mathtt{Bot} \Uparrow^V \mathtt{Bot} \qquad \text{(VU-Bot)}$$

$$\frac{\mathtt{X} \in V \qquad , \vdash ,(\mathtt{X}) \Uparrow^V \mathtt{T}}{, \vdash \mathtt{X} \Uparrow^V \mathtt{T}} \qquad \text{(VU-Var-1)}$$

$$\frac{\mathtt{X} \notin V}{, \vdash \mathtt{X} \Uparrow^V \mathtt{X}} \qquad \text{(VU-Var-2)}$$

$$\frac{FV(\overline{\mathtt{A}}) \cap V = \emptyset \qquad ,,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}} \vdash \overline{\mathtt{S}} \Downarrow_V \overline{\mathtt{S}}' \qquad ,,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}} \vdash \mathtt{T} \Uparrow^V \mathtt{T}'}{, \vdash \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}\rightarrow\mathtt{T} \Uparrow^V \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}'\rightarrow\mathtt{T}'} \qquad \text{(VU-Fun-1)}$$

$$\frac{FV(\overline{\mathtt{A}}) \cap V \neq \emptyset}{, \vdash \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}\rightarrow\mathtt{T} \Uparrow^V \mathtt{Top}} \qquad \text{(VU-Fun-2)}$$

The definition of $, \vdash \mathtt{S} \Downarrow_V \mathtt{T}$ is similar to $, \vdash \mathtt{S} \Uparrow^V \mathtt{T}$:

$$, \vdash \mathtt{Top} \Downarrow_V \mathtt{Top} \qquad \text{(VD-Top)}$$

$$, \vdash \mathtt{Bot} \Downarrow_V \mathtt{Bot} \qquad \text{(VD-Bot)}$$

$$\frac{\mathtt{X} \in V}{, \vdash \mathtt{X} \Downarrow_V \mathtt{Bot}} \qquad \text{(VD-Var-1)}$$

$$\frac{\mathtt{X} \notin V}{, \vdash \mathtt{X} \Downarrow_V \mathtt{X}} \qquad \text{(VD-Var-2)}$$

$$\frac{FV(\overline{\mathtt{A}}) \cap V = \emptyset \qquad \Gamma, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{A}} \vdash \overline{\mathtt{S}} \Uparrow^V \overline{\mathtt{S}'} \qquad \Gamma, \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{A}} \vdash \mathtt{T} \Downarrow_V \mathtt{T}'}{\Gamma \vdash \mathtt{All}(\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}\to\mathtt{T} \Downarrow_V \mathtt{All}(\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}'}\to\mathtt{T}'} \quad \text{(VD-Fun-1)}$$

$$\frac{FV(\overline{\mathtt{A}}) \cap V \neq \emptyset}{\Gamma \vdash \mathtt{All}(\overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}\to\mathtt{T} \Downarrow_V \mathtt{Bot}} \quad \text{(VD-Fun-2)}$$

It is easy to check that, for each variable set V, $\Uparrow^V$ and $\Downarrow_V$ are total functions. (These functions are similar to the ones used in [GP97], but somewhat simpler because of the presence of `Bot` in our type system.)

3.2.1 Lemma [Soundness of variable elimination]:

1. If $\Gamma \vdash \mathtt{S} \Uparrow^V \mathtt{T}$ then $FV(\mathtt{T}) \cap V = \emptyset$ and $\Gamma \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{S} \Downarrow_V \mathtt{T}$ then $FV(\mathtt{T}) \cap V = \emptyset$ and $\Gamma \vdash \mathtt{T} \texttt{ <: } \mathtt{S}$.

Proof: By a straightforward simultaneous induction on variable-elimination derivations. ∎

3.2.2 Lemma [Completeness of variable elimination]:

1. If $\Gamma \vdash \mathtt{S} \texttt{ <: } \mathtt{T}$ and $FV(\mathtt{T}) \cap V = \emptyset$, then $\Gamma \vdash \mathtt{S} \Uparrow^V \mathtt{R}$ with $\Gamma \vdash \mathtt{R} \texttt{ <: } \mathtt{T}$.
2. If $\Gamma \vdash \mathtt{T} \texttt{ <: } \mathtt{S}$ and $FV(\mathtt{T}) \cap V = \emptyset$, then $\Gamma \vdash \mathtt{S} \Downarrow_V \mathtt{R}$ with $\Gamma \vdash \mathtt{T} \texttt{ <: } \mathtt{R}$.

Proof: See [Pie97]. ∎

3.3 Constraints

Next, we introduce the constraints that will be manipulated by our algorithm. We need constraints of two forms, one for recording the fact that a type variable `X` must be exactly equal to some type `T` (for example, `X` must be exactly equal to `Bot` in order to make `All(Y<:X)Y→Y` a subtype of `All(Y<:Bot)Y→Y`), and the other for recording the fact that a variable `X` must lie between two types `S` and `T` (for example, `X` must lie between `A` and `B` in order to make `X→X` a subtype of `A→B`).

Formally, a *V-constraint* has one of the forms

$[\mathtt{T}]$	equality constraint
$[\mathtt{S},\mathtt{T}]$	subtyping constraint

with the additional constraint that all the free variables of `S` and `T` are distinct from V. A type `R` *satisfies* a constraint c, written $\Gamma \vdash \mathtt{R} \in c$, if

$$\begin{array}{llll} & c = [\mathtt{S}] & \text{and} & \mathtt{R} = \mathtt{S} \\ \text{or} & c = [\mathtt{S},\mathtt{T}] & \text{and} & \Gamma \vdash \mathtt{S} \texttt{ <: } \mathtt{R} \quad \text{and} \quad \Gamma \vdash \mathtt{R} \texttt{ <: } \mathtt{T}. \end{array}$$

The *maximal* and *minimal* types satisfying a given constraint are defined in the obvious way:

$$\begin{array}{lllllll} \max([\mathtt{S},\mathtt{T}]) & = & \mathtt{T} & \qquad & \max([\mathtt{S}]) & = & \mathtt{S} \\ \min([\mathtt{S},\mathtt{T}]) & = & \mathtt{S} & & \min([\mathtt{S}]) & = & \mathtt{S} \end{array}$$

An $\overline{\mathtt{X}}/V$*-constraint set* C is a finite map from $\overline{\mathtt{X}}$ to V-constraints. The empty $\overline{\mathtt{X}}/V$-constraint set, written $\emptyset$, maps each variable $\mathtt{X}_i$ to the constraint $[\mathtt{Bot},\mathtt{Top}]$. The singleton $\overline{\mathtt{X}}/V$-constraint set $\{\mathtt{X}_i \mapsto c\}$ maps $\mathtt{X}_i$ to the constraint c and every other $\mathtt{X}_j$ to $[\mathtt{Bot},\mathtt{Top}]$. The *meet* of two V-constraints is defined as follows (for all cases other than those specified below, the meet is undefined):

$$\begin{array}{llll} [\mathtt{S}] \wedge [\mathtt{S}] & = & [\mathtt{S}] & \\ [\mathtt{S}] \wedge [\mathtt{U},\mathtt{V}] & = & [\mathtt{S}] & \text{if } \Gamma \vdash \mathtt{U} \texttt{ <: } \mathtt{S} \texttt{ <: } \mathtt{V} \\ [\mathtt{S},\mathtt{T}] \wedge [\mathtt{U}] & = & [\mathtt{U}] & \text{if } \Gamma \vdash \mathtt{S} \texttt{ <: } \mathtt{U} \texttt{ <: } \mathtt{T} \\ [\mathtt{S},\mathtt{T}] \wedge [\mathtt{U},\mathtt{V}] & = & [\mathtt{J},\mathtt{M}] & \text{if } \Gamma \vdash \mathtt{S} \vee \mathtt{U} = \mathtt{J} \text{ and } \Gamma \vdash \mathtt{T} \wedge \mathtt{V} = \mathtt{M} \end{array}$$

The meet operation is extended pointwise to constraint sets.

$$(C \wedge D)(\mathtt{X}_i) = C(\mathtt{X}_i) \wedge D(\mathtt{X}_i)$$

We write $\overline{C} \wedge \overline{D}$ to abbreviate $C_1 \wedge \ldots \wedge C_m \wedge D_1 \wedge \ldots \wedge D_n$.

3.4 Constraint Generation

Our constraint generation rules have the form

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{T} \Rightarrow C$$

and define a partial function that, given a typing context Γ, a set of type variables V, a set of unknowns $\overline{\mathtt{X}}$, and two types S and T, calculates the minimal $\overline{\mathtt{X}}/V$-constraint set C guaranteeing that $\Gamma \vdash \mathtt{S} <: \mathtt{T}$.

The set V allows us to avoid generating nonsensical constraint sets in which bound variables are mentioned outside their scopes (this part of the constraint generation problem is similar to *mixed-prefix unification* [Mil92]). For example, if we are interested in constraining X so that `All(Y<:Top)()`$\rightarrow$`(Y`$\rightarrow$`Y)` is a subtype of `All(Y<:Top)()`$\rightarrow$`X`, we should not return the constraint set $\{\mathtt{X}\mapsto[\mathtt{Y}\rightarrow\mathtt{Y},\mathtt{Top}]\}$, since Y would be out of scope. Instead, we should return the constraint set $\{\mathtt{X}\mapsto[\mathtt{Bot}\rightarrow\mathtt{Top},\mathtt{Top}]\}$, which is in fact the weakest constraint on X guaranteeing that `All(Y<:Top)()`$\rightarrow$`(Y`$\rightarrow$`Y)` is a subtype of `All(Y<:Top)()`$\rightarrow$`X`.

Our constraint generation algorithm is defined by the following collection of rules. In the definition, we suppose that $\overline{\mathtt{X}} \cap V = \emptyset$. More importantly, we assume (and recursively maintain) the invariant that only one of S and T mentions the variables $\overline{\mathtt{X}}$ (i.e. either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$). This is crucial to the completeness of the algorithm, since it ensures we only have to solve a matching problem (modulo subtyping) rather than a unification problem.

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{T} <: \mathtt{Top} \Rightarrow \emptyset$$

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Bot} <: \mathtt{T} \Rightarrow \emptyset$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad \mathtt{T} \Downarrow_V \mathtt{R}}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} <: \mathtt{T} \Rightarrow \{\mathtt{Y}\mapsto[\mathtt{Bot},\mathtt{R}]\}}$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad \mathtt{T} \Uparrow^V \mathtt{R}}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{T} <: \mathtt{Y} \Rightarrow \{\mathtt{Y}\mapsto[\mathtt{R},\mathtt{Top}]\}}$$

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} <: \mathtt{Y} \Rightarrow \emptyset$$

$$\frac{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \Gamma(\mathtt{Y}) <: \mathtt{T} \Rightarrow C}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} <: \mathtt{T} \Rightarrow C}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \overline{\mathtt{A}} \equiv \overline{\mathtt{B}} \Rightarrow \overline{\mathtt{K}}, \overline{D} \qquad V' = V \cup \{\overline{\mathtt{Y}}\} \\ \overline{\mathtt{Y}} \cap V = \emptyset \qquad \overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset \qquad FV(\sigma) \cap \overline{\mathtt{Y}} = \emptyset \\ \Gamma, \overline{\mathtt{Y}}<:\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \overline{\mathtt{T}} <: \overline{\mathtt{R}} \Rightarrow \overline{C} \qquad \Gamma, \overline{\mathtt{Y}}<:\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{U} \Rightarrow D\end{array}}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}}<:\overline{\mathtt{A}})\overline{\mathtt{R}}\rightarrow\mathtt{S} <: \mathtt{All}(\overline{\mathtt{Y}}<:\overline{\mathtt{B}})\overline{\mathtt{T}}\rightarrow\mathtt{U} \Rightarrow D \wedge \overline{D} \wedge \overline{C}}$$

In the clause for quantifiers (whose bounds must match exactly rather than modulo subtyping), we need an auxiliary "matching relation" $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \mathtt{U}, C$, which yields both a constraint set C whose solutions make S and T identical and a type U that is equal to whichever of S and T is concrete (recall that the variables $\overline{\mathtt{X}}$ do not occur in one of S or T). The definition of this relation follows the same lines as the main constraint generator:

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Top} \equiv \mathtt{Top} \Rightarrow \mathtt{Top}, \emptyset$$

$$\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Bot} \equiv \mathtt{Bot} \Rightarrow \mathtt{Bot}, \emptyset$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad V \cap FV(\mathtt{T}) = \emptyset}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} \equiv \mathtt{T} \Rightarrow \mathtt{T}, \{\mathtt{Y}\mapsto[\mathtt{T}]\}}$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad V \cap FV(\mathtt{T}) = \emptyset}{\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{T} \equiv \mathtt{Y} \Rightarrow \mathtt{T}, \{\mathtt{Y}\mapsto[\mathtt{T}]\}}$$

$$\frac{\mathtt{Y} \notin \overline{\mathtt{X}}}{,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} \equiv \mathtt{Y} \Rightarrow \mathtt{Y}, \emptyset}$$

$$\frac{\begin{array}{c} ,\ \vdash^{V}_{\overline{\mathtt{X}}} \overline{\mathtt{A}} \equiv \overline{\mathtt{B}} \Rightarrow \overline{\mathtt{K}}, \overline{D} \qquad V' = V \cup \{\overline{\mathtt{Y}}\} \\ \overline{\mathtt{Y}} \cap V = \emptyset \qquad \overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset \qquad FV(\sigma) \cap \overline{\mathtt{Y}} = \emptyset \\ ,\ ,\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \overline{\mathtt{T}} \equiv \overline{\mathtt{S}} \Rightarrow \overline{\mathtt{L}}, \overline{C} \qquad ,\ ,\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \mathtt{U} \equiv \mathtt{V} \Rightarrow \mathtt{M}, D \end{array}}{,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{A}})\overline{\mathtt{R}}\to\mathtt{S} \equiv \mathtt{All}(\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{B}})\overline{\mathtt{T}}\to\mathtt{U} \Rightarrow \mathtt{All}(\overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{K}})\overline{\mathtt{L}}\to\mathtt{M},\ D \wedge \overline{D} \wedge \overline{C}}$$

Note that the C returned by these algorithms is always an $\overline{\mathtt{X}}/V$-constraint set. Also, if $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S}$ <: $\mathtt{T} \Rightarrow C$ and the variables $\overline{\mathtt{X}}$ do not appear in $\mathtt{S}$ or $\mathtt{T}$, then the constraint set C is always empty. The constraint generator in this case is effectively just the subtyping relation.

3.5 Soundness and Completeness of Constraint Generation

Each constraint set returned by the constraint generator characterizes a collection of substitutions associating concrete types with the names of the missing type parameters. Formally, a *substitution* is a finite map from type variables to types. More precisely, an $\overline{\mathtt{X}}/V$-substitution is one whose domain is $\overline{\mathtt{X}}$ and whose codomain contains types whose free variables are distinct from V. We write $\sigma[\mathtt{X}_i \mapsto \mathtt{T}]$ for the substitution that behaves like σ everywhere except at $\mathtt{X}_i$, where its value is $\mathtt{T}$.

We say that an $\overline{\mathtt{X}}/V$-substitution σ *satisfies* an $\overline{\mathtt{X}}/V$-constraint set C, written $,\ \vdash \sigma \in C$, iff $,\ \vdash \sigma(\mathtt{X}_i) \in C(\mathtt{X}_i)$ for each variable $\mathtt{X}_i$. A constraint set is called *satisfiable* if there is some substitution that satisfies it. (Note that this condition can be checked very easily, by verifying that $,\ \vdash \min(C(\mathtt{X}_i))$ <: $\max(C(\mathtt{X}_i))$ for each $\mathtt{X}_i$.)

If C and D are two $\overline{\mathtt{X}}/V$-constraint sets such that $,\ \vdash \sigma \in C$ implies $,\ \vdash \sigma \in D$ for all σ, we say that C is *more demanding than* D. Note that the meet of constraint sets defined above yields a greatest lower bound in this ordering and that the empty constraint set is maximal (i.e., least demanding).

Before we can prove soundness and completeness for the constraint generator, we need analogous lemmas for the auxiliary "matching constraint" generator.

3.5.1 Lemma [Soundness of matching constraint generation]: Suppose that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. If $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \mathtt{U}, C$ and $\sigma \in C$, then $\sigma\mathtt{S} = \sigma\mathtt{T} = \mathtt{U}$.

Proof: Straightforward induction. ■

3.5.2 Lemma [Completeness of matching constraint generation]: Let σ be an $\overline{\mathtt{X}}/V$-substitution with $\overline{\mathtt{X}} \cap V = \emptyset$, and let $\mathtt{S}$ and $\mathtt{T}$ be types such that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. If $\sigma\mathtt{S} = \sigma\mathtt{T}$, then $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \sigma\mathtt{S}, C$ for some C such that $,\ \vdash \sigma \in C$.

Proof: By induction on the structure of $\sigma\mathtt{S}$ ($= \sigma\mathtt{T}$). We only give the most interesting cases of the proof. The remaining cases follow easily using the induction hypothesis and, for the function case, the fact that $,\ \vdash \sigma \in C$ and $,\ \vdash \sigma \in D$ implies $,\ \vdash \sigma \in C \wedge D$.

Case: $\mathtt{S} = \mathtt{Y}$ where $\mathtt{Y} \in \overline{\mathtt{X}}$

It must be the case that $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$, since $\mathtt{Y} \in \overline{\mathtt{X}}$. We therefore have $\sigma\mathtt{Y} = \sigma\mathtt{T} = \mathtt{T}$. It must also be the case that $FV(\mathtt{T}) \cap V = \emptyset$, since $\mathtt{T}$ occurs in the codomain of σ and σ is a $\overline{\mathtt{X}}/V$-substitution. We therefore have $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \mathtt{T}, \{\mathtt{Y}\mapsto[\mathtt{T}]\}$ and $,\ \vdash \sigma \in \{\mathtt{Y}\mapsto[\mathtt{T}]\}$ as required.

Case: $\mathtt{T} = \mathtt{Y}$ where $\mathtt{Y} \in \overline{\mathtt{X}}$

Similar. ■

3.5.3 Proposition [Soundness of constraint generation]: Suppose that $FV(,\ ) \cap \overline{\mathtt{X}} = \emptyset$, that $dom(,\ ) \cap \overline{\mathtt{X}} = \emptyset$, and that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. If $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S}$ <: $\mathtt{T} \Rightarrow C$ and $,\ \vdash \sigma \in C$, then $,\ \vdash \sigma\mathtt{S}$ <: $\sigma\mathtt{T}$.

Proof: By induction on the derivation of $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S}$ <: $\mathtt{T} \Rightarrow C$.

Case: $,\ \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S}$ <: $\mathtt{Top} \Rightarrow \emptyset$

Immediate, since $\sigma\mathtt{Top} = \mathtt{Top}$ and , $\vdash \sigma\mathtt{S}$ <: $\mathtt{Top}$, no matter what $\sigma\mathtt{S}$ is.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Bot}$ <: $\mathtt{T} \Rightarrow \emptyset$

Immediate, since $\sigma\mathtt{Bot} = \mathtt{Bot}$ and , $\vdash \mathtt{Bot}$ <: $\sigma\mathtt{T}$ no matter what $\sigma\mathtt{T}$ is.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Y}$ <: $\mathtt{T} \Rightarrow C$ where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\mathtt{T} \Downarrow_V \mathtt{R}$ and $C = \{\mathtt{Y} \mapsto [\mathtt{Bot},\mathtt{R}]\}$

Since , $\vdash \sigma \in C$ we have , $\vdash \sigma\mathtt{Y}$ <: $\mathtt{R}$. Since $\mathtt{Y} \in \overline{\mathtt{X}}$, we know that $\sigma\mathtt{T} = \mathtt{T}$; also, Lemma 3.2.1(2) tells us that , $\vdash \mathtt{R}$ <: $\mathtt{T}$. We therefore have , $\vdash \sigma\mathtt{Y}$ <: $\mathtt{R}$ <: $\mathtt{T} = \sigma\mathtt{T}$, as required.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{S}$ <: $\mathtt{Y} \Rightarrow C$ where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\mathtt{S} \Uparrow^V \mathtt{R}$ and $C = \{\mathtt{Y} \mapsto [\mathtt{R},\mathtt{Top}]\}$

Since , $\vdash \sigma \in C$ we have , $\vdash \mathtt{R}$ <: $\sigma\mathtt{Y}$. Since $\mathtt{Y} \in \overline{\mathtt{X}}$, we know that $\sigma\mathtt{S} = \mathtt{S}$; also, Lemma 3.2.1(1) tells us that , $\vdash \mathtt{S}$ <: $\mathtt{R}$. We therefore have , $\vdash \sigma\mathtt{S} = \mathtt{S}$ <: $\mathtt{R}$ <: $\sigma\mathtt{Y}$, as required.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Y}$ <: $\mathtt{Y} \Rightarrow \emptyset$

Since, by assumption, the variables $\overline{\mathtt{X}}$ do not appear free in both $\mathtt{S}$ and $\mathtt{T}$, it must be the case that $\mathtt{Y} \notin \overline{\mathtt{X}}$. Thus, $\sigma\mathtt{Y} = \mathtt{Y}$ and , $\vdash \mathtt{Y}$ <: $\mathtt{Y}$, as required.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Y}$ <: $\mathtt{T} \Rightarrow C$ where , $\vdash^{V}_{\overline{\mathtt{X}}}$, $(\mathtt{Y})$ <: $\mathtt{T} \Rightarrow C$

Using the induction hypothesis, we obtain , $\vdash \sigma($, $(\mathtt{Y}))$ <: $\sigma\mathtt{T}$. Since $\mathtt{Y} \in \mathit{dom}($, $)$, we know that $\mathtt{Y} \notin \overline{\mathtt{X}}$. The fact that $FV($, $(\mathtt{Y})) \cap \overline{\mathtt{X}} = \emptyset$ follows from our assumption that $FV($, $) \cap \overline{\mathtt{X}} = \emptyset$. We therefore have , $\vdash$, $(\mathtt{Y})$ <: $\sigma\mathtt{T}$. Using the S-Var rule, we obtain , $\vdash \mathtt{Y}$ <: $\sigma\mathtt{T}$, which is what we need since $\sigma\mathtt{Y} = \mathtt{Y}$.

Case: , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{All}(\overline{\mathtt{Y}}$<:$\overline{\mathtt{A}})\overline{\mathtt{R}}{\rightarrow}\mathtt{S}$ <: $\mathtt{All}(\overline{\mathtt{Y}}$<:$\overline{\mathtt{B}})\overline{\mathtt{T}}{\rightarrow}\mathtt{U} \Rightarrow D \wedge \overline{D} \wedge \overline{C}$ where , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\overline{\mathtt{A}} \equiv \overline{\mathtt{B}} \Rightarrow \overline{\mathtt{K}}, \overline{D}$ and ,$\overline{\mathtt{Y}}$<:$\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}}$ $\overline{\mathtt{T}}$ <: $\overline{\mathtt{R}} \Rightarrow \overline{C}$ and ,$\overline{\mathtt{Y}}$<:$\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}}$ $\mathtt{S}$ <: $\mathtt{U} \Rightarrow D$ and $V' = V \cup \{\overline{\mathtt{Y}}\}$ and $\overline{\mathtt{Y}} \cap V = \emptyset$ and $\overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset$ and $FV(\sigma) \cap \overline{\mathtt{Y}} = \emptyset$

The side conditions on $\overline{\mathtt{Y}}$ imply that σ is a valid $\overline{\mathtt{X}}/V'$-substitution. Therefore, our assumption that , $\vdash \sigma \in D \wedge \overline{D} \wedge \overline{C}$, plus the fact that $FV(\overline{\mathtt{K}}) \cap \overline{\mathtt{X}} = \emptyset$, is sufficient to allow us to use the induction hypothesis to prove that ,$\overline{\mathtt{Y}}$<:$\overline{\mathtt{K}} \vdash \sigma\mathtt{S}$ <: $\sigma\mathtt{U}$ and ,$\overline{\mathtt{Y}}$<:$\overline{\mathtt{K}} \vdash \sigma\overline{\mathtt{T}}$ <: $\sigma\overline{\mathtt{R}}$. Moreover, Lemma 3.5.1 tells us that $\sigma\overline{\mathtt{A}} = \sigma\overline{\mathtt{B}} = \overline{\mathtt{K}}$. By the subtyping rule for functions, we conclude that , $\vdash \mathtt{All}(\overline{\mathtt{Y}}$<:$\sigma\overline{\mathtt{A}})\sigma\overline{\mathtt{R}}{\rightarrow}\sigma\mathtt{S}$ <: $\mathtt{All}(\overline{\mathtt{Y}}$<:$\sigma\overline{\mathtt{B}})\sigma\overline{\mathtt{T}}{\rightarrow}\sigma\mathtt{U}$. The result now follows, since $FV(\sigma) = \emptyset$ ensures that $\mathtt{All}(\overline{\mathtt{Y}}$<:$\sigma\overline{\mathtt{A}})\sigma\overline{\mathtt{R}}{\rightarrow}\sigma\mathtt{S} = \sigma(\mathtt{All}(\overline{\mathtt{Y}}$<:$\overline{\mathtt{A}})\overline{\mathtt{R}}{\rightarrow}\mathtt{S})$ and $\mathtt{All}(\overline{\mathtt{Y}}$<:$\sigma\overline{\mathtt{B}})\sigma\overline{\mathtt{T}}{\rightarrow}\sigma\mathtt{U} = \sigma(\mathtt{All}(\overline{\mathtt{Y}}$<:$\overline{\mathtt{B}})\overline{\mathtt{T}}{\rightarrow}\mathtt{U})$. ∎

3.5.4 Proposition [Completeness of constraint generation]: Let σ be an $\overline{\mathtt{X}}/V$-substitution with $\overline{\mathtt{X}} \cap V = \emptyset$, and let $\mathtt{S}$ and $\mathtt{T}$ be types such that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. Let , be a context such that $\overline{\mathtt{X}} \cap \mathit{dom}($, $) = \emptyset$ and $FV($, $) \cap \overline{\mathtt{X}} = \emptyset$. If , $\vdash \sigma\mathtt{S}$ <: $\sigma\mathtt{T}$, then , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{S}$ <: $\mathtt{T} \Rightarrow C$ and , $\vdash \sigma \in C$.

Proof: By induction on the depth of a derivation of , $\vdash \sigma\mathtt{S}$ <: $\sigma\mathtt{T}$.

Case: $\mathtt{S} = \mathtt{Y}$ where $\mathtt{Y} \in \overline{\mathtt{X}}$

We have , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Y}$ <: $\mathtt{T} \Rightarrow C$ where $C = \{\mathtt{Y} \mapsto [\mathtt{Bot},\mathtt{R}]\}$ and $\mathtt{T} \Downarrow_V \mathtt{R}$. Now, since σ is a $\overline{\mathtt{X}}/V$-substitution, we know that $FV(\sigma\mathtt{Y}) \cap V = \emptyset$, and therefore, using Lemma 3.2.2, we have , $\vdash \sigma\mathtt{Y}$ <: $\mathtt{R}$. This ensures that , $\vdash \sigma \in \mathtt{C}$, as required.

Case: $\mathtt{T} = \mathtt{Y}$ where $\mathtt{Y} \in \overline{\mathtt{X}}$

We have , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{S}$ <: $\mathtt{Y} \Rightarrow C$ where $C = \{\mathtt{Y} \mapsto [\mathtt{R},\mathtt{Top}]\}$ and $\mathtt{S} \Uparrow^V \mathtt{R}$. Now, since σ is a $\overline{\mathtt{X}}/V$-substitution, we know that $FV(\sigma\mathtt{Y}) \cap V = \emptyset$, and therefore, using Lemma 3.2.2, we have , $\vdash \mathtt{R}$ <: $\sigma\mathtt{Y}$. This ensures that , $\vdash \sigma \in \mathtt{C}$, as required.

Case: $\mathtt{T} = \mathtt{Top}$

Immediate, since , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{S}$ <: $\mathtt{Top} \Rightarrow \emptyset$ and , $\vdash \sigma \in \emptyset$.

Case: $\mathtt{S} = \mathtt{Bot}$

Immediate, since , $\vdash^{V}_{\overline{\mathtt{X}}}$ $\mathtt{Bot}$ <: $\mathtt{T} \Rightarrow \emptyset$ and , $\vdash \sigma \in \emptyset$.

Case: $\mathtt{S} = \mathtt{Y}$ and $\mathtt{T} = \mathtt{Y}$ where $\mathtt{Y} \notin \overline{\mathtt{X}}$

Immediate, since , $\vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{Y} \Rightarrow \emptyset$ and $\sigma \in \emptyset$.

Case: , $\vdash \mathtt{Y} \mathrel{<:} \sigma\mathtt{T}$ where , $\vdash$, $(\mathtt{Y}) \mathrel{<:} \sigma\mathtt{T}$

Since $FV(,) \cap \overline{\mathtt{X}} = \emptyset$ we have $\sigma(,(\mathtt{Y})) =$, $(\mathtt{Y})$, so we can use the induction hypothesis to prove that , $\vdash^{V}_{\overline{\mathtt{X}}}$, $(\mathtt{Y}) \mathrel{<:} \mathtt{T} \Rightarrow C$ and , $\vdash \sigma \in C$. The result follows directly, since , $\vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{T} \Rightarrow C$.

Case: , $\vdash \mathtt{All}(\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{A}})\sigma\overline{\mathtt{R}}\to\sigma\mathtt{S} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{B}})\sigma\overline{\mathtt{T}}\to\sigma\mathtt{U}$, where $\sigma\overline{\mathtt{A}} = \sigma\overline{\mathtt{B}}$ and , $,\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{B}} \vdash \sigma\overline{\mathtt{T}} \mathrel{<:} \sigma\overline{\mathtt{R}}$ and , $,\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{B}} \vdash \sigma\mathtt{S} \mathrel{<:} \sigma\mathtt{U}$

Since we identify type expressions up to alpha-conversion, we may suppose (wlog) that the $\overline{\mathtt{Y}}$ are chosen so that $\overline{\mathtt{Y}}\cap V = \emptyset$, $\overline{\mathtt{Y}}\cap\overline{\mathtt{X}} = \emptyset$, and $FV(\sigma)\cap\overline{\mathtt{Y}} = \emptyset$. Thus, σ is a valid $\overline{\mathtt{X}}/V'$-substitution and we can use the induction hypothesis to prove that , $,\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{B}} \vdash^{V'}_{\overline{\mathtt{X}}} \overline{\mathtt{T}} \mathrel{<:} \overline{\mathtt{R}} \Rightarrow \overline{C}$ and , $\vdash \sigma \in \overline{C}$, and similarly, , $,\overline{\mathtt{Y}}\mathrel{<:}\sigma\overline{\mathtt{B}} \vdash^{V'}_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{U} \Rightarrow D$ and , $\vdash \sigma \in D$. Using Lemma 3.5.2, we have that , $\vdash^{V}_{\overline{\mathtt{X}}} \overline{\mathtt{A}} \equiv \overline{\mathtt{B}} \Rightarrow \sigma\overline{\mathtt{B}}, \overline{D}$ and , $\vdash \sigma \in \overline{D}$. So, by the constraint generation rule for function types, we have , $\vdash^{V}_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}}\mathrel{<:}\overline{\mathtt{A}})\overline{\mathtt{S}}\to\mathtt{P} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Y}}\mathrel{<:}\overline{\mathtt{B}})\overline{\mathtt{T}}\to\mathtt{Q} \Rightarrow D\wedge\overline{D}\wedge\overline{C}$. Finally, by the fact that $D\wedge\overline{D}\wedge\overline{C}$ is a greatest lower bound, we have , $\vdash \sigma \in D\wedge\overline{D}\wedge\overline{C}$, as required. ∎

3.6 Calculating Type Arguments

Having generated a set of constraints for the missing type parameters $\overline{\mathtt{X}}$, the final job of the local constraint solver is to choose values for $\overline{\mathtt{X}}$ that make the result type of the whole application as small as possible. Depending on where the variables $\overline{\mathtt{X}}$ occur in $\mathtt{R}$, this may involve choosing the smallest possible values for some variables and the largest for others. For example, if $\mathtt{R} = \mathtt{X}\to\mathtt{Y}$ and we have generated the constraint set $\{\mathtt{X} \mapsto [\mathtt{S},\mathtt{T}], \mathtt{Y} \mapsto [\mathtt{U},\mathtt{V}]\}$, the smallest possible value for $\mathtt{R}$ is found by maximizing $\mathtt{X}$ and minimizing $\mathtt{Y}$—i.e., by taking the substitution $\sigma = [\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{U}]$. It may also be the case that no substitution for the variables yields a minimal result type; for example, if $\mathtt{R} = \mathtt{X}\to\mathtt{X}$ and we have the constraint set $\{\mathtt{X} \mapsto [\mathtt{Int},\mathtt{Top}]\}$, then both $\mathtt{Int}\to\mathtt{Int}$ and $\mathtt{Top}\to\mathtt{Top}$ are solutions but neither is a subtype of the other. Local type argument synthesis fails in this case (as required by the specification in Section 3.1).

We begin by formalizing the ways in which maximizing or minimizing $\mathtt{X}$ affect the final result type.

1. We say that $\mathtt{R}$ is *covariant in* $\mathtt{X}$ if , $\vdash [\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ whenever , $\vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.
2. We say that $\mathtt{R}$ is *contravariant in* $\mathtt{X}$ if , $\vdash [\mathtt{T}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{S}/\mathtt{X}]\mathtt{R}$ whenever , $\vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.
3. We say that $\mathtt{R}$ is *invariant in* $\mathtt{X}$ if , $\vdash [\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ only when both , $\vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ and , $\vdash \mathtt{T} \mathrel{<:} \mathtt{S}$.
4. We say that $\mathtt{R}$ is *rigid in* $\mathtt{X}$ if , $\vdash [\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ only when $\mathtt{S} = \mathtt{T}$.

It is easy to check whether $\mathtt{R}$ is covariant, contravariant, invariant, or rigid in a given variable $\mathtt{X}$ by examining where $\mathtt{X}$ occurs in $\mathtt{R}$ (to the right or left of arrows, in the bounds of type binders, etc.).

Next, we need a technical definition characterizing types whose equivalence classes in the subtype relation are singletons. For example, if $\mathtt{X}\mathrel{<:}\mathtt{Bot}$, then $\mathtt{X}\to\mathtt{X}$ is equivalent, but not identical, to $\mathtt{Bot}\to\mathtt{Bot}$: indeed its equivalence class has several members. On the other hand, $\mathtt{Top}$ is only equivalent to itself. Formally, we call a type variable a *bottom variable* if it is bounded by $\mathtt{Bot}$ or by another bottom variable. Now, let , be a context and $\mathtt{S}$ a type whose free variables are in $dom(,)$. We say that $\mathtt{S}$ is *rigid under* , if

- $\mathtt{S} = \mathtt{Top}$;
- $\mathtt{S} = \mathtt{Bot}$ and no variable in , is bounded by $\mathtt{Bot}$;
- $\mathtt{S} = \mathtt{X}$ and $\mathtt{X}$ is not a bottom variable;
- $\mathtt{S} = \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{S}}\to\mathtt{T}$ with each $\mathtt{S}_i$ rigid under , $,\mathtt{X}_1{:}\mathtt{S}_1,\ldots,\mathtt{X}_{i-1}{:}\mathtt{S}_{i-1}$ and $\mathtt{T}$ rigid under , $,\overline{\mathtt{X}}{:}\overline{\mathtt{S}}$;

Extending the notion of rigidity from types to constraints, we say that a ,-constraint c is *rigid* if it admits only one solution—i.e., if either $c = [\mathtt{S}]$ or else $c = [\mathtt{S},\mathtt{S}]$, where $\mathtt{S}$ is rigid under , . Similarly, c is said to be *tight* if it admits only one solution, up to equivalence—i.e., if either $c = [\mathtt{S}]$ or else $c = [\mathtt{S},\mathtt{T}]$ with both , $\vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ and , $\vdash \mathtt{T} \mathrel{<:} \mathtt{S}$.

3.6.1 Lemma: If $\mathtt{S}$ is rigid under , , then every type equivalent to $\mathtt{S}$ is syntactically equal to $\mathtt{S}$—i.e., , $\vdash \mathtt{S} <: \mathtt{T}$ and , $\vdash \mathtt{T} <: \mathtt{S}$ together imply that $\mathtt{S}$ and $\mathtt{T}$ are identical.

Proof: See [Pie97, Lemma 4.1.2]. ■

3.6.2 Corollary: If c is rigid under , and , $\vdash \mathtt{S} \in c$ and , $\vdash \mathtt{T} \in c$, then $\mathtt{S}$ and $\mathtt{T}$ are identical.

With the forgoing definitions in hand, we can now show how to choose values for the variables $\overline{\mathtt{X}}$ that will minimize $\mathtt{R}$ (or else determine that this is not possible). Let C be a satisfiable $\overline{\mathtt{X}}/V$-constraint set and $\mathtt{R}$ a type whose free variables are in $dom(,) \cup \{\overline{\mathtt{X}}\}$. Let the substitution $\sigma_{C\mathtt{R}}$ be defined (when it exists) as follows:

For each $\mathtt{X}_i$...
  if $\mathtt{R}$ is covariant in $\mathtt{X}_i$,
    then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \min(C(\mathtt{X}_i))$
  else if $\mathtt{R}$ is contravariant in $\mathtt{X}_i$,
    then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \max(C(\mathtt{X}_i))$
  else if $\mathtt{R}$ is invariant in $\mathtt{X}_i$
    and $C(\mathtt{X}_i)$ is tight,
    then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \min(C(\mathtt{X}_i))$
  else if $\mathtt{R}$ is rigid in $\mathtt{X}_i$,
    and $C(\mathtt{X}_i)$ is rigid,
    then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \min(C(\mathtt{X}_i))$
  else $\sigma_{C\mathtt{R}}$ is undefined.

It remains to verify that the substitution $\sigma_{C\mathtt{R}}$ chosen in this way is indeed the best possible. Let C be an $\overline{\mathtt{X}}/V$-constraint set, and σ be a $\overline{\mathtt{X}}/V$-substitution. We say that σ is *minimal* for C and $\mathtt{R}$, written , $\vdash \sigma \in C \Downarrow \mathtt{R}$, if , $\vdash \sigma \in C$ and, for all $\overline{\mathtt{X}}/V$-substitutions σ' with , $\vdash \sigma' \in C$, we have , $\vdash \sigma\mathtt{R} <: \sigma'\mathtt{R}$.

3.6.3 Proposition:

1. If the substitution $\sigma_{C\mathtt{R}}$ exists, then it is a minimal substitution for C and $\mathtt{R}$.

2. If $\sigma_{C\mathtt{R}}$ is undefined, then C and $\mathtt{R}$ have no minimal substitution.

Proof:

1. Suppose $\sigma_{C\mathtt{R}}$ exists, and that σ' is another substitution such that , $\vdash \sigma' \in C$. We must show that , $\vdash \sigma_{C\mathtt{R}}\mathtt{R} <: \sigma'\mathtt{R}$.

   Let $n = |\overline{\mathtt{X}}|$, and construct a sequence of substitutions $\sigma_0, \ldots, \sigma_n$ as follows:

$$\begin{array}{rcll} \sigma_0 & = & \sigma_{C\mathtt{R}} & \\ \sigma_i & = & \sigma_{i-1}[\mathtt{X}_i \mapsto \sigma'(\mathtt{X}_i)] & \text{if } i \geq 1. \end{array}$$

   Note that $\sigma_n = \sigma'$. We now argue that , $\vdash \sigma_{i-1}\mathtt{R} <: \sigma_i\mathtt{R}$ for each $i \geq 1$.

   - If $\mathtt{R}$ is covariant in $\mathtt{X}_i$, then, by definition, $\sigma_{i-1}\mathtt{X}_i = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \min(C(\mathtt{X}_i))$, and thus , $\vdash \sigma_{i-1}(\mathtt{X}_i) <: \sigma_i(\mathtt{X}_i)$. But this implies that , $\vdash \sigma_{i-1}\mathtt{R} <: \sigma_i\mathtt{R}$, by the definition of covariance.
   - Similarly, if $\mathtt{R}$ is contravariant in $\mathtt{X}_i$, then $\sigma_{i-1}\mathtt{X}_i = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \max(C(\mathtt{X}_i))$, and thus , $\vdash \sigma_i(\mathtt{X}_i) <: \sigma_{i-1}(\mathtt{X}_i)$, which implies that , $\vdash \sigma_{i-1}\mathtt{R} <: \sigma_i\mathtt{R}$, by the definition of contravariance.
   - If $\mathtt{R}$ is invariant in $\mathtt{X}_i$, then $\sigma_{i-1}\mathtt{X}_i = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \max(C(\mathtt{X}_i))$, and we also know that , $\vdash \min(C(\mathtt{X}_i)) <: \max(C(\mathtt{X}_i)) <: \min(C(\mathtt{X}_i))$. But since , $\vdash \min(C(\mathtt{X}_i)) <: \sigma_i(\mathtt{X}_i)) <: \min(C(\mathtt{X}_i))$, we have by transitivity that , $\vdash \sigma_i(\mathtt{X}_i) <: \sigma_{i-1}(\mathtt{X}_i) <: \sigma_i(\mathtt{X}_i)$, which, by the definition of invariance, yields , $\vdash \sigma_{i-1}\mathtt{R} <: \sigma_i\mathtt{R}$.
   - Finally, if $\mathtt{R}$ is rigid in $\mathtt{X}_i$, then $\sigma_{i-1}(\mathtt{X}_i) = \sigma_i(\mathtt{X}_i)$, and so , $\vdash \sigma_{i-1}\mathtt{R} <: \sigma_i\mathtt{R}$ by reflexivity of subtyping.

We have thus shown that $, \vdash \sigma_{C\mathtt{R}}\mathtt{R} = \sigma_0\mathtt{R} \mathrel{<:} \sigma_1\mathtt{R} \mathrel{<:} \cdots \mathrel{<:} \sigma_n\mathtt{R} = \sigma'\mathtt{R}$, and the desired result follows by transitivity of subtyping.

2. If $\sigma_{C\mathtt{R}}$ is undefined, then either C is unsatisfiable (in which case the result holds trivially) or else C is satisfiable and we must show that no substitution that satisfies it is minimal. So suppose, for a contradiction, that σ is minimal for C and $\mathtt{R}$. There are two cases to consider, depending on why $\sigma_{C\mathtt{R}}$ failed to be defined:

   (a) For some $\mathtt{X}_i$, $\mathtt{R}$ is invariant in $\mathtt{X}_i$ but $C(\mathtt{X}_i)$ is not a tight constraint. In this case, we know that there must be some $\mathtt{T}$ such that $, \vdash \mathtt{T} \in C(\mathtt{X}_i)$ but such that either $, \vdash \sigma(\mathtt{X}_i) \not\mathrel{<:} \mathtt{T}$ or $, \vdash \mathtt{T} \not\mathrel{<:} \sigma(\mathtt{X}_i)$. We can then construct a substitution $\sigma' = [\mathtt{X}_i \mapsto \mathtt{T}]$ such that $, \vdash \sigma' \in C$ and, since $\mathtt{X}_i$ is invariant in $\mathtt{R}$, such that $, \vdash \sigma\mathtt{R} \not\mathrel{<:} \sigma'\mathtt{R}$, contradicting our assumption that σ is minimal for C and $\mathtt{R}$.

   (b) For some $\mathtt{X}_i$, $\mathtt{R}$ is rigid in $\mathtt{X}_i$ but $C(\mathtt{X}_i)$ is not a rigid constraint. In this case, we know that there must be some $\mathtt{T}$ different from $\sigma(\mathtt{X}_i)$ such that $, \vdash \mathtt{T} \in C(\mathtt{X}_i)$. We can then construct a substitution $\sigma' = \sigma[\mathtt{X}_i \mapsto \mathtt{T}]$ such that $, \vdash \sigma' \in C$ and, since $\mathtt{X}_i$ in rigid in $\mathtt{R}$, such that $, \vdash \sigma\mathtt{R} \not\mathrel{<:} \sigma'\mathtt{R}$, contradicting our assumption that σ is minimal for C and $\mathtt{R}$. ∎

3.6.4 Corollary: The algorithmic rule

$$\frac{\begin{array}{c} , \vdash \mathtt{f} \in \mathtt{F} \qquad , \vdash \mathtt{F} \Uparrow \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{S}})\overline{\mathtt{T}}\mathtt{\to R} \\ , \vdash \overline{\mathtt{e}} \in \overline{\mathtt{U}} \qquad |\overline{\mathtt{X}}| > 0 \qquad \overline{\mathtt{X}} \cap FV(\overline{\mathtt{S}}) = \emptyset \\ , \vdash^{\emptyset}_{\overline{\mathtt{X}}} \overline{\mathtt{A}} \mathrel{<:} \overline{\mathtt{S}} \Rightarrow \overline{C} \qquad , \vdash^{\emptyset}_{\overline{\mathtt{X}}} \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{T}} \Rightarrow \overline{D} \qquad , \vdash \sigma \in (\overline{C} \wedge \overline{D}) \Downarrow \mathtt{R} \end{array}}{, \vdash \mathtt{f}(\overline{\mathtt{e}}) \in \sigma\mathtt{R} \Rightarrow \mathtt{f}[\sigma\overline{\mathtt{X}}](\overline{\mathtt{e}})}$$

is equivalent to the declarative rule given in Section 3.1.

Acknowledgements

The paper was mostly written while Turner was visiting Indiana University in the summer of '97. Pierce was partially supported by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects.*

References

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CMMS94] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. A preliminary version appeared in TACS '91 (Sendai, Japan, pp. 750–770).

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[Ghe90] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Università di Pisa, March 1990. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

[GP97] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 1997. To appear.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, October 1992.

[Pie97] Benjamin C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998. To appear. Full version available as Indiana University CSCI technical report #493.