Local Type Inference

BENJAMIN C. PIERCE
University of Pennsylvania
and
DAVID N. TURNER
An Teallach, Ltd.

We study two partial type inference methods for a language combining subtyping and impredicative polymorphism. Both methods are *local* in the sense that missing annotations are recovered using only information from adjacent nodes in the syntax tree, without long-distance constraints such as unification variables. One method infers type arguments in polymorphic applications using a local constraint solver. The other infers annotations on bound variables in function abstractions by propagating type constraints downward from enclosing application nodes. We motivate our design choices by a statistical analysis of the uses of type inference in a sizable body of existing ML code.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms: Languages, Theory

Additional Key Words and Phrases: Polymorphism, subtyping, type inference

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 $F_{\leq}$ [Cardelli and Wegner 1985; Curien and Ghelli 1992; Cardelli et al. 1994], but these features have not yet been satisfactorily

This is a revised and extended version of a paper presented at the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1998. Most of the paper was written while Turner was visiting Indiana University in Summer '97. Pierce was supported by Indiana University and the University of Pennsylvania, and by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects.*
Authors' addresses: B.C. Pierce, Department of Computer and Information Science, Univ. of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104-6389; email bcpierce@cis.upenn.edu; D.N. Turner, An Teallach, Ltd., Technology Transfer Center, King's Buildings, Edinburgh, EH9 3JL, UK; email dnt@cruach.freeserve.co.uk.

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 this article, we pursue a much simpler 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.

Our goal is to develop simple, well-behaved type inference techniques for new language designs in the style of Quest [Cardelli 1991], Pizza [Odersky and Wadler 1997], GJ [Bracha et al. 1998] or ML2000—designs supporting both object-oriented programming idioms and the characteristic coding styles of languages such as ML and Haskell. In particular, we shall use the shorthand *ML-style programming* to refer to a style in which (1) the use of higher-order functions and anonymous abstractions is encouraged; (2) polymorphic definitions are used freely and at a fairly fine grain (for individual function definitions rather than whole modules); and (3) "pure" data structures are used instead of mutable state, whenever possible. Our goal might then be restated as "type inference for ML-style programming in the presence of subtyping."

In particular, we are concerned with languages whose type-theoretic core combines subtyping and impredicative polymorphism in the style of System F [Girard 1972; Reynolds 1974]. This combination of features places us in the realm of partial type inference methods, since complete type inference for impredicative polymorphism alone is already known to be undecidable [Wells 1994], and the addition of subtyping does not seem to make the problem any easier. (For the combination of subtyping with Hindley/Milner-style polymorphic type inference, promising results have been reported [Aiken and Wimmers 1993; Eifrig et al. 1995; Jagannathan and Wright 1995; Trifonov and Smith 1996; Sulzmann et al. 1997; Flanagan and Felleisen 1997; Pottier 1997], but practical checkers based on these results have yet to see widespread use.)

1.1 How Much Inference Is Enough?

The job of a partial type inference algorithm should be to eliminate especially those type annotations that are both *common* and *silly*—i.e., those that can be neither justified on the basis of their value as checked documentation nor ignored because they are rare.

Unfortunately, each of the characteristic features of ML-style (polymorphic instantiation, anonymous function abstractions, and pure data structures) does give rise to a certain number of silly annotations that would not be required if the same program were expressed in a first-order, imperative style. To get a rough idea of the actual numbers, we made some simple measurements of a sizable body of existing code—about 160,000 lines of ML, written by several different programming teams. The results of these measurements can be summarized as follows (they are reported in detail in Appendix A):

—Polymorphic instantiation (i.e., type application) is ubiquitous, occurring in every third line of code, on average.

—Anonymous function definitions occur anywhere from once per 10 lines to once per 100 lines of code, depending on style.

—The manipulation of pure data structures leads to many local variable bindings (occurring, on average, once every 12 lines). However, in all but one of the programs we measured, local definitions of functions only occur once in 66 lines.

These observations give a fairly clear indication of the properties that a type inference scheme should have in order to support the ML programming style conveniently:

(1) To make fine-grained polymorphism tolerable, type arguments in applications of polymorphic functions must usually be inferred. However, it is acceptable to require annotations on the bound variables of top-level function definitions (since these usually provide useful documentation) and local function definitions (since these are relatively rare).

(2) To make higher-order programming convenient, it is helpful, though not absolutely necessary, to infer the types of parameters to anonymous function definitions.

(3) To support the manipulation of pure data structures, local bindings should not usually require explicit annotations.

Note that, even though we have motivated our design choices by an analysis of ML programming styles, it is not our intention to provide the same degree of type inference as is possible in languages based on Hindley-Milner polymorphism. Rather, we want to exchange complete type inference for simpler methods that work well in the presence of more powerful type-theoretic features such as subtyping and impredicative polymorphism.

1.2 Local Type Inference

In this article, we propose two specific partial type inference techniques that, together, satisfy all three of the requirements listed above.

(1) An algorithm for *local synthesis of type arguments* that infers the "locally best possible" values for types omitted from polymorphic applications whenever such best values exist. The expected and actual types of the term arguments are compared to yield a set of subtyping constraints on the missing type arguments; their values are then selected so as to satisfy these constraints while making the result type of the whole application as informative (small) as possible.

(2) *Bidirectional propagation* of type information allows the types of parameters of anonymous functions to be inferred. When an anonymous function appears as an argument to another function, the expected domain type is used as the expected type for the anonymous abstraction, allowing the type annotations on its parameters to be omitted. A similar, but even simpler, technique infers type annotations on local variable bindings.

Both of these methods are *local*, in the sense that type information is propagated only between adjacent nodes in the syntax tree. Indeed, their simplicity—and, in the case of type argument synthesis, its completeness relative to a simple declarative specification—rests on this property.

The basic idea of bidirectional checking is well known as folklore. Similar ideas have been used, for example, in ML compilers and typecheckers based on attribute grammars. However, this technique has usually been combined with ML-style type inference (see, for example, Aditya and Nikhil [1991]); it is surprisingly powerful when used by itself as a local type inference method. Specific technical contributions of this article are the formalization of bidirectional checking in a setting with both subtyping and impredicative polymorphism and the combination of this idea with the technique for local synthesis of type arguments presented in the previous section.

The remainder of the article is organized as follows. In the next section, we define a fully typed internal language. Sections 3, 4, and 5 develop the techniques of local synthesis of type arguments and bidirectional checking in detail, first for (in Sections 3 and 4) a simplified language with subtyping and unbounded universal polymorphism, then (in Section 5) extending this treatment to bounded quantifiers. Section 6 sketches some possible extensions. Section 7 surveys related work. Section 8 offers evaluation and concluding remarks. Details of our measurements of ML programs appear in an appendix.

Some additional experiments with using local type inference in practice are reported in Hosoya and Pierce [1999].

2. INTERNAL LANGUAGE

When discussing type inference, it is useful to think of a statically typed language in three parts:

(1) Syntax, typing rules, and semantics for a fully typed *internal language*.

(2) An *external language* in which some type annotations are made optional or omitted entirely. This is the language that the programmer actually uses. (In some programming languages, the internal and external language may differ in more than just type annotations, and type inference may perform nontrivial transformations on program structure. For example, under certain assumptions ML's generic `let`-definition mechanism can be viewed in this way.)

(3) Some specification of a *type inference* relation between the external language and the internal one. (The terms *type inference*, *type reconstruction*, and *type synthesis* have all been used for this relation. We choose "inference" as the most generic.)

In explicitly typed languages, the external and internal forms are essentially the same, and the type reconstruction relation is the identity. In implicitly typed languages, the external language allows all type annotations to be omitted, and type reconstruction promises to fill in all missing type information. On the other hand, we can describe a language as *partially typed* if the internal and external forms are not the same, but the specification of type inference does not guarantee that omitted annotations can always be inferred.[1]

[1]Another possible sense of the phrase *partial type inference* occurs when the specification of type reconstruction is only partially implementable: the language definition promises to infer more than the compiler can actually do. We reject this definition, since it underspecifies the type inference algorithm, allowing different compilers to use different heuristics and leading to unportable programs.

Our internal language—the target for the type inference methods described in Sections 3 and 4—is based on the language Kernel $F_{\leq}$, Cardelli and Wegner's core calculus of subtyping and impredicative polymorphism. We consider first a simplified fragment of the full system, in which variables are all unbounded (i.e., all quantifiers are of the form `All(X)T`, not `All(X<:S)T`). The treatment here will be extended to deal with bounded quantifiers in Section 5, but the simple language presented first is enough to show all of the essential ideas and the technical development is easier to follow.

2.1 Syntax

Besides the restriction to unbounded quantifiers, we modify the usual definition of System $F_{\leq}$ [Cardelli and Wegner 1985] in two significant ways. First, we add a minimal type `Bot`. As we shall see in Section 3, 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. (`Bot` is also an interesting typing feature in its own right: for example, it can be used as the result type of non-returning expressions such as exception-raising primitives.[2])

Second, 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 does not change the expressiveness of the language, but 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:

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

We use the metavariables `R`, `S`, `T`, `U`, and `V` to range over types; `e` and `f` range over terms. We use the notation $\overline{\texttt{X}}$ to denote the sequence $\texttt{X}_1,\ldots,\texttt{X}_n$, and similarly $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$ to denote $\texttt{x}_1\texttt{:T}_1,\ldots,\texttt{x}_n\texttt{:T}_n$. We write $\Gamma(\texttt{x})$ for the type of `x` in Γ.

[2]It is worth noting that, even without such primitives, `Bot` changes the set of typeable terms of the language. For example, the untyped term `fun(x) x (x+1)` can be typed as `fun(x:Bot) x (x+1)`.

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, and judgments that differ only in the names of bound variables are regarded as identical. Binders in contexts are assumed to have distinct names. The rules for scoping of bound variables are as usual (in $\mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}\to\mathtt{T}$, the variables $\overline{\mathtt{X}}$ are in scope in $\overline{\mathtt{S}}$ and $\mathtt{T}$). $FV(\mathtt{T})$, the set of type variables free in $\mathtt{T}$, is defined in the usual way. We write $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}$ for the simultaneous sustitution of $\overline{\mathtt{T}}$ for $\overline{\mathtt{X}}$ in $\mathtt{S}$.

2.2 Subtyping

Our subtyping relation is quite simple because of the restriction to unbounded quantification. In particular, the addition of the bottom type Bot in this context is straightforward. We write $\overline{\mathtt{S}}$ <: $\overline{\mathtt{T}}$ to mean "$|\overline{\mathtt{S}}| = |\overline{\mathtt{T}}|$ and $\mathtt{S}_i$ <: $\mathtt{T}_i$ for all $1 \leq i \leq |\overline{\mathtt{S}}|$."

$$\mathtt{X} <: \mathtt{X} \tag{S-Refl}$$

$$\mathtt{T} <: \mathtt{Top} \tag{S-Top}$$

$$\mathtt{Bot} <: \mathtt{T} \tag{S-Bot}$$

$$\frac{\overline{\mathtt{T}} <: \overline{\mathtt{R}} \qquad \mathtt{S} <: \mathtt{U}}{\mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{R}}\to\mathtt{S} <: \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{T}}\to\mathtt{U}} \tag{S-Fun}$$

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:

LEMMA 2.2.1 (TRANSITIVITY). *If* S <: T *and* T <: U *then* S <: U.

PROOF. A simple induction on the derivations of S <: T and T <: U. The cases involving Top and Bot rely on the fact that R <: Bot implies R = Bot, and Top <: R implies R = Top. □

LEMMA 2.2.2 (REFLEXIVITY). T <: T*, for all* T.

PROOF. A simple induction on the structure of T. □

We use the notation $\mathtt{S} \vee \mathtt{T}$ to denote the least upper bound of S and T, and $\mathtt{S} \wedge \mathtt{T}$ for the greatest lower bound of S and T.

$$\mathtt{S} \vee \mathtt{T} = \begin{cases} \mathtt{T} & \text{if } \mathtt{S} \mathrel{<:} \mathtt{T} \\ \mathtt{S} & \text{if } \mathtt{T} \mathrel{<:} \mathtt{S} \\ \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{M}}{\rightarrow}\mathtt{J} & \text{if } \mathtt{S} = \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{V}}{\rightarrow}\mathtt{P} \\ & \quad \mathtt{T} = \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{W}}{\rightarrow}\mathtt{Q} \\ & \quad \overline{\mathtt{V}} \wedge \overline{\mathtt{W}} = \overline{\mathtt{M}} \\ & \quad \mathtt{P} \vee \mathtt{Q} = \mathtt{J} \\ \mathtt{Top} & \text{otherwise} \end{cases}$$

$$\mathtt{S} \wedge \mathtt{T} = \begin{cases} \mathtt{S} & \text{if } \mathtt{S} \mathrel{<:} \mathtt{T} \\ \mathtt{T} & \text{if } \mathtt{T} \mathrel{<:} \mathtt{S} \\ \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{J}}{\rightarrow}\mathtt{M} & \text{if } \mathtt{S} = \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{V}}{\rightarrow}\mathtt{P} \\ & \quad \mathtt{T} = \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{W}}{\rightarrow}\mathtt{Q} \\ & \quad \overline{\mathtt{V}} \vee \overline{\mathtt{W}} = \overline{\mathtt{J}} \\ & \quad \mathtt{P} \wedge \mathtt{Q} = \mathtt{M} \\ \mathtt{Bot} & \text{otherwise} \end{cases}$$

Note that $\wedge$ and $\vee$ are total functions: for every Γ, $\mathtt{S}$, and $\mathtt{T}$, there are unique types $\mathtt{M}$ and $\mathtt{J}$ such that $\mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ and $\mathtt{S} \vee \mathtt{T} = \mathtt{J}$. It is easy to check that these definitions have the appropriate universal properties:

Lemma 2.2.3.

(1) $\mathtt{S} \mathrel{<:} (\mathtt{S} \vee \mathtt{T})$ *and* $\mathtt{T} \mathrel{<:} (\mathtt{S} \vee \mathtt{T})$.

(2) $(\mathtt{S} \wedge \mathtt{T}) \mathrel{<:} \mathtt{S}$ *and* $(\mathtt{S} \wedge \mathtt{T}) \mathrel{<:} \mathtt{T}$.

Proof. We prove both parts simultaneously, using induction on the structure of $\mathtt{S}$ and $\mathtt{T}$. □

Lemma 2.2.4.

(1) If $\mathtt{S} \mathrel{<:} \mathtt{U}$ *and* $\mathtt{T} \mathrel{<:} \mathtt{U}$ *then* $(\mathtt{S} \vee \mathtt{T}) \mathrel{<:} \mathtt{U}$.

(2) If $\mathtt{U} \mathrel{<:} \mathtt{S}$ *and* $\mathtt{U} \mathrel{<:} \mathtt{T}$ *then* $\mathtt{U} \mathrel{<:} (\mathtt{S} \wedge \mathtt{T})$.

Proof. We prove both parts simultaneously, using induction on the structure of $\mathtt{U}$. □

2.3 Explicit Typing Rules

The typing relation $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ 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 *unique type* (sometimes called the *manifest type* of the term), corresponding to its minimal type in the system with subsumption. Note that this stylistic choice does not change the set of typable terms—just the number of typing derivations showing that a given term is typable.

The typing rule for variables is standard.

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(Var)}$$

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

$$\frac{\Gamma, \overline{\mathtt{X}}, \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e} \in \mathtt{T}}{\Gamma \vdash \mathtt{fun[}\overline{\mathtt{X}}\mathtt{]\,(}\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}\mathtt{)e} \in \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}\to\mathtt{T}} \qquad \text{(Abs)}$$

Similarly, the rule for applications combines the usual application and polymorphic application rules. We calculate the type of the function and check that the provided term and type arguments are consistent with the function type. The result type of the application is found by substituting the actual type arguments into the function's result type.

$$\frac{\Gamma \vdash \mathtt{f} \in \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}\to\mathtt{R} \qquad \Gamma \vdash \overline{\mathtt{e}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}}{\Gamma \vdash \mathtt{f\,[}\overline{\mathtt{T}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)} \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{R}} \qquad \text{(App)}$$

$\Gamma \vdash \overline{\mathtt{e}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$ here is an abbreviation for "$\Gamma \vdash \overline{\mathtt{e}} \mathrel{<:} \overline{\mathtt{U}}$ and $\overline{\mathtt{U}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$."

To finish the definition of the typing relation, another rule is required. To see why, note that Bot <: $\mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}\to\mathtt{T}$ for any $\overline{\mathtt{X}}$, $\overline{\mathtt{S}}$, 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):

$$\frac{\Gamma \vdash \mathtt{f} \in \mathtt{Bot} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{f\,[}\overline{\mathtt{T}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{Bot}} \qquad \text{(App-Bot)}$$

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

Theorem 2.3.1 (Uniqueness of Manifest Types). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{S}$ *and* $\Gamma \vdash \mathtt{e} \in \mathtt{T}$*, then* $\mathtt{S} = \mathtt{T}$.

The definitions of operational and denotational semantics for the internal language 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

In the introduction, we identified three categories of type annotations that are worth inferring automatically: type arguments in applications of polymorphic functions, annotations on bound variables in anonymous function abstractions, and annotations on local variable bindings. In this section, we address the first of these, leaving the second and third for Section 4.

Our measurements of ML programs (presented in the appendix) 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. We therefore believe that it is essential to have some form of type argument synthesis in any language intended to support

ML-style programming. For example, consider the polymorphic identity function `id` with type `All(X)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)`.

When considering the general problem of type argument synthesis, the first question we have to answer is: How do we decide where type arguments have been omitted (and therefore need to be synthesized)? In the variant of $F_{\leq}$ we presented in Section 2, the answer is simple: we look for application nodes where the function is polymorphic but there are no explicit type arguments. For example, the fact that `id` is polymorphic makes it clear that a type argument is missing in the application `id(3)`. (An alternative approach is to require an explicit marker at each point where a type argument is missing. We did not pursue this scheme, since marking all the positions where a type argument is required can be quite cumbersome. However, some of the partial type inference schemes proposed by Pfenning [1988a] have used this scheme, with additional refinements which allow the type argument markers themselves to be elided.)

The second 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 application. For example, both `id[Int](x)` and `id[Real](x)` are valid completions of the term `id(x)`, where $\mathtt{x} \in \mathtt{Int}$ 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)()→(X→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→Int` and `Real→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 Section 4 we show how propagating additional contextual information sometimes allows us to avoid this situation).

3.1 Specification

The syntax of the external language is identical to that of the internal language, since external-language applications can already be written without type arguments (using our convention that zero-length lists of type arguments can be omitted entirely). All we need to do is to define a four-place *type inference* relation:

$$\Gamma \vdash \mathsf{e} \in \mathtt{T} \Rightarrow \mathsf{e}'$$

Intuitively, this relation can be read "In context Γ, type annotations can be added to the external language term `e` to yield the internal language term e', which has type `T`."

The specification of the type inference relation is quite simple. For each typing rule in the internal language with conclusion $\Gamma \vdash \mathsf{e}' \in \mathtt{T}$, the type inference relation contains an analogous rule with conclusion $\Gamma \vdash \mathsf{e} \in \mathtt{T} \Rightarrow \mathsf{e}'$, where 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:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \in \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{T}}\to\mathtt{R} \Rightarrow \mathtt{f}' \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \Rightarrow \overline{\mathtt{e}}' \qquad |\overline{\mathtt{X}}| > 0 \qquad \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \\ \forall\overline{\mathtt{V}}.\ (\overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \text{ implies } [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{R} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{X}}]\mathtt{R})\end{array}}{\Gamma \vdash \mathtt{f}(\overline{\mathtt{e}}) \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{R} \Rightarrow \mathtt{f}'[\overline{\mathtt{U}}](\overline{\mathtt{e}}')} \qquad \text{(App-InfSpec)}$$

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 do not 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.)

The type arguments $\overline{\mathtt{U}}$ that we pick in the conclusion of our synthesis rule must satisfy a number of conditions. Firstly, the types of the value parameters $\overline{\mathtt{S}}$ must be subtypes of the function's parameter types $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$. Secondly, the arguments $\overline{\mathtt{U}}$ must be chosen in such a way that any other choice of arguments $\overline{\mathtt{V}}$ satisfying the previous condition will yield a less informative result type, i.e., a supertype of $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{R}$.

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).

THEOREM 3.1.1 (SOUNDNESS). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}'$, *then* `e` *is a partial erasure of* $\mathtt{e}'$ *and* $\Gamma \vdash \mathtt{e}' \in \mathtt{T}$.

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, 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]

It should be emphasized that the App-InfSpec rule (together with the rest of the rules for the typing relation of the internal 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 supertype (or subtype) in which these variables do not occur. Formally, we write $\mathtt{S} \Uparrow^{V} \mathtt{T}$ for

[3]When we extend the system to include bounded type quantification in Section 5, this straightforward completeness property will be weakened a little, since we do not presently know how to infer type arguments for multiple quantifiers with interdependent bounds.

the relation "`T` is the least supertype of `S` such that $FV(\mathtt{T}) \cap V = \emptyset$" and $\mathtt{S} \Downarrow^V \mathtt{T}$ for the dual relation "`T` is the greatest subtype of `S` such that $FV(\mathtt{T}) \cap V = \emptyset$." Fortunately, such types can always be found. For example, suppose $V = \{\mathtt{X}\}$; then $\mathtt{(X,Int)}{\rightarrow}\mathtt{X} \Uparrow^V \mathtt{(Bot,Int)}{\rightarrow}\mathtt{Top}$.

The variable-elimination-by-promotion relation can be computed as follows:

$$\mathtt{Top} \Uparrow^V \mathtt{Top} \tag{VU-Top}$$

$$\mathtt{Bot} \Uparrow^V \mathtt{Bot} \tag{VU-Bot}$$

$$\frac{\mathtt{X} \in V}{\mathtt{X} \Uparrow^V \mathtt{Top}} \tag{VU-Var-1}$$

$$\frac{\mathtt{X} \notin V}{\mathtt{X} \Uparrow^V \mathtt{X}} \tag{VU-Var-2}$$

$$\frac{\overline{\mathtt{S}} \Downarrow^V \overline{\mathtt{U}} \qquad \mathtt{T} \Uparrow^V \mathtt{R} \qquad \overline{\mathtt{X}} \notin V}{\mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{T} \Uparrow^V \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{U}}{\rightarrow}\mathtt{R}} \tag{VU-Fun}$$

The relation $\mathtt{S} \Downarrow^V \mathtt{T}$ is defined analogously:

$$\mathtt{Top} \Downarrow^V \mathtt{Top} \tag{VD-Top}$$

$$\mathtt{Bot} \Downarrow^V \mathtt{Bot} \tag{VD-Bot}$$

$$\frac{\mathtt{X} \in V}{\mathtt{X} \Downarrow^V \mathtt{Bot}} \tag{VD-Var-1}$$

$$\frac{\mathtt{X} \notin V}{\mathtt{X} \Downarrow^V \mathtt{X}} \tag{VD-Var-2}$$

$$\frac{\overline{\mathtt{S}} \Uparrow^V \overline{\mathtt{U}} \qquad \mathtt{T} \Downarrow^V \mathtt{R} \qquad \overline{\mathtt{X}} \notin V}{\mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{T} \Downarrow^V \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{U}}{\rightarrow}\mathtt{R}} \tag{VD-Fun}$$

It is easy to check that $\Uparrow^V$ and $\Downarrow^V$ are total functions, for any given set V. (These functions are similar to the ones used in Ghelli and Pierce [1998], but somewhat simpler because of the presence of `Bot` in our type system.)

Lemma 3.2.1 (Soundness).

(*1*) *If* $\mathtt{S} \Uparrow^V \mathtt{T}$ *then* $FV(\mathtt{T}) \cap V = \emptyset$ *and* `S <: T`.
(*2*) *If* $\mathtt{S} \Downarrow^V \mathtt{T}$ *then* $FV(\mathtt{T}) \cap V = \emptyset$ *and* `T <: S`.

Proof. A simple simultaneous induction on the variable-elimination derivations. □

Lemma 3.2.2 (Completeness).

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

(*2*) *If* $\mathtt{T} <: \mathtt{S}$ *and* $FV(\mathtt{T}) \cap V = \emptyset$, *then* $\mathtt{S} \Downarrow^V \mathtt{R}$ *with* $\mathtt{T} <: \mathtt{R}$.

Proof. A simple simultaneous induction on the subtype derivations, using the fact that, for all R, $\mathtt{X} <: \mathtt{R}$ implies $\mathtt{R} = \mathtt{X}$ or $\mathtt{R} = \mathtt{Top}$, and $\mathtt{R} <: \mathtt{X}$ implies $\mathtt{R} = \mathtt{Bot}$ or $\mathtt{R} = \mathtt{X}$. □

3.3 Constraint Generation

Next, we introduce the constraint sets that will be manipulated by our algorithm. Each constraint has the form $\mathtt{S}_i <: \mathtt{X}_i <: \mathtt{T}_i$, recording a lower and upper bound for $\mathtt{X}_i$. An $\overline{\mathtt{X}}/V$*-constraint set* C has the form

$$\{\mathtt{S}_i <: \mathtt{X}_i <: \mathtt{T}_i \mid (FV(\mathtt{S}_i) \cup FV(\mathtt{T}_i)) \cap (V \cup \overline{\mathtt{X}}) = \emptyset\}.$$

The empty $\overline{\mathtt{X}}/V$-constraint set, written $\emptyset$, contains the trivial constraint $\mathtt{Bot} <: \mathtt{X}_i <: \mathtt{Top}$ for each variable $\mathtt{X}_i$. The singleton $\overline{\mathtt{X}}/V$-constraint set $\{\mathtt{S} <: \mathtt{X}_i <: \mathtt{T}\}$ includes the constraint $\mathtt{S} <: \mathtt{X}_i <: \mathtt{T}$ for $\mathtt{X}_i$ and trivial constraints for every other $\mathtt{X}_j$. The *meet* of two $\overline{\mathtt{X}}/V$-constraints C and D, written $C \wedge D$, is defined as follows:

$$\{\mathtt{S} \vee \mathtt{U} <: \mathtt{X}_i <: \mathtt{T} \wedge \mathtt{V} \mid \mathtt{S} <: \mathtt{X}_i <: \mathtt{T} \in C \text{ and } \mathtt{U} <: \mathtt{X}_i <: \mathtt{V} \in D\}$$

We write $\bigwedge \overline{C}$ to abbreviate $C_1 \wedge \ldots \wedge C_n$.

Our constraint generation rules have the form

$$V \vdash_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{T} \Rightarrow C$$

and define a partial function that, given a set of type variables V, a set of unknowns $\overline{\mathtt{X}}$, and two types S and T, calculates the *minimal* (i.e., least constraining) $\overline{\mathtt{X}}/V$-constraint set C that guarantees $\mathtt{S} <: \mathtt{T}$.

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* [Miller 1992]). For example, if we are interested in constraining X so that $\mathtt{All(Y)()}\to\mathtt{(Y}\to\mathtt{Y)}$ is a subtype of $\mathtt{All(Y)()}\to\mathtt{X}$, we should not return the constraint set $\{\mathtt{Y}\to\mathtt{Y} <: \mathtt{X} <: \mathtt{Top}\}$, since Y would be out of scope. Instead, we should return the constraint set $\{\mathtt{Bot}\to\mathtt{Top} <: \mathtt{X} <: \mathtt{Top}\}$, which is in fact the weakest constraint on X guaranteeing that $\mathtt{All(Y)()}\to\mathtt{(Y}\to\mathtt{Y)}$ is a subtype of $\mathtt{All(Y)()}\to\mathtt{X}$.

Our constraint generation algorithm is defined by the following collection of rules, where we always suppose that $\overline{\mathtt{X}} \cap V = \emptyset$.

$$V \vdash_{\overline{\mathtt{X}}} \mathtt{T} <: \mathtt{Top} \Rightarrow \emptyset \tag{CG-Top}$$

$$V \vdash_{\overline{\mathtt{X}}} \mathtt{Bot} <: \mathtt{T} \Rightarrow \emptyset \tag{CG-Bot}$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad \mathtt{S} \Downarrow^V \mathtt{T} \qquad FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset}{V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} <: \mathtt{S} \Rightarrow \{\mathtt{Bot} <: \mathtt{Y} <: \mathtt{T}\}} \tag{CG-Upper}$$

$$\frac{\mathtt{Y} \in \overline{\mathtt{X}} \qquad \mathtt{S} \Uparrow^V \mathtt{T} \qquad FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset}{V \vdash_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{Y} \Rightarrow \{\mathtt{T} <: \mathtt{Y} <: \mathtt{Top}\}} \tag{CG-Lower}$$

$$\frac{\mathtt{Y} \notin \overline{\mathtt{X}}}{V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{Y} \Rightarrow \emptyset} \qquad \text{(CG-Refl)}$$

$$\frac{V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \overline{\mathtt{T}} \mathrel{<:} \overline{\mathtt{R}} \Rightarrow \overline{C} \qquad V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{U} \Rightarrow D \qquad \overline{\mathtt{Y}} \cap (V \cup \overline{\mathtt{X}}) = \emptyset}{V \vdash_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{R}} {\rightarrow} \mathtt{S} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{T}} {\rightarrow} \mathtt{U} \Rightarrow (\bigwedge \overline{C}) \wedge D} \qquad \text{(CG-Fun)}$$

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

When we "call" the constraint generator in a statement of the form $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{T} \Rightarrow C$, it will always be the case that only one of $\mathtt{S}$ and $\mathtt{T}$ 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 our constraint-solving method, since it ensures we only have to solve a matching-modulo-subtyping problem rather than a unification-modulo-subtyping problem.

3.4 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. An *$\overline{\mathtt{X}}/V$-substitution* σ is a finite map from type variables to types whose domain is $\overline{\mathtt{X}}$ with $FV(\sigma\mathtt{X}_i) \cap V = \emptyset$ for all $\mathtt{X}_i$. 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 $\mathtt{T}$.

Suppose σ is an $\overline{\mathtt{X}}/V$-substitution and $\overline{\mathtt{X}} \cap V = \emptyset$. We say that σ *satisfies* an $\overline{\mathtt{X}}/V$-constraint set C, written $\sigma \in C$, if $\mathtt{S}_i \mathrel{<:} \sigma(\mathtt{X}_i) \mathrel{<:} \mathtt{T}_i$ for each $(\mathtt{S}_i \mathrel{<:} \mathtt{X}_i \mathrel{<:} \mathtt{T}_i) \in C$.[4] A constraint set is *satisfiable* if there is some substitution that satisfies it. Note that this condition can be checked very easily, by verifying that $\mathtt{S}_i \mathrel{<:} \mathtt{T}_i$ for each $(\mathtt{S}_i \mathrel{<:} \mathtt{X}_i \mathrel{<:} \mathtt{T}_i) \in C$.

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

Proposition 3.4.1 (Soundness). *If $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{T} \Rightarrow C$ and $\sigma \in C$, then $\sigma\mathtt{S} \mathrel{<:} \sigma\mathtt{T}$.*

Proof. By induction on the derivation of $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{T} \Rightarrow C$.

Case CG-Top: $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{Top} \Rightarrow \emptyset$. Immediate, since $\sigma\mathtt{Top} = \mathtt{Top}$ and $\sigma\mathtt{S} \mathrel{<:} \mathtt{Top}$ for all $\sigma\mathtt{S}$.

Case CG-Bot: $V \vdash_{\overline{\mathtt{X}}} \mathtt{Bot} \mathrel{<:} \mathtt{T} \Rightarrow \emptyset$. Similar: since $\sigma\mathtt{Bot} = \mathtt{Bot}$ and $\mathtt{Bot} \mathrel{<:} \sigma\mathtt{T}$ for all $\sigma\mathtt{T}$.

Case CG-Upper: $V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{T} \Rightarrow \{\mathtt{Bot} \mathrel{<:} \mathtt{Y} \mathrel{<:} \mathtt{R}\}$, where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\mathtt{T} \Downarrow^V \mathtt{R}$, and $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. Since $\sigma \in C$ we have $\sigma\mathtt{Y} \mathrel{<:} \mathtt{R}$. Using Lemma 3.2.1 we have $\mathtt{R} \mathrel{<:} \mathtt{T}$. Since $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$ we have that $\sigma\mathtt{T} = \mathtt{T}$ and $\sigma\mathtt{Y} \mathrel{<:} \sigma\mathtt{T}$ as required.

[4]An alternative, somewhat more standard, definition would be "$\sigma \in C$ iff $\sigma\mathtt{S}_i \mathrel{<:} \sigma(\mathtt{X}_i) \mathrel{<:} \sigma\mathtt{T}_i$ for each $(\mathtt{S}_i \mathrel{<:} \mathtt{X}_i \mathrel{<:} \mathtt{T}_i) \in C$." We prefer our formulation, since it emphasizes the fact that the $\mathtt{X}$s do not occur at all in the upper or lower bounds.

Case CG-Lower: $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{Y} \Rightarrow \{\mathtt{R} \mathrel{<:} \mathtt{Y} \mathrel{<:} \mathtt{Top}\}$, where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\mathtt{S} \Uparrow^V \mathtt{R}$, and $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$. Since $\sigma \in C$ we have $\mathtt{R} \mathrel{<:} \sigma\mathtt{Y}$. Using Lemma 3.2.1 we have $\mathtt{S} \mathrel{<:} \mathtt{R}$. Since $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ we have that $\sigma\mathtt{S} = \mathtt{S}$ and $\sigma\mathtt{S} \mathrel{<:} \sigma\mathtt{Y}$ as required.

Case CG-Refl: $V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{Y} \Rightarrow \emptyset$, where $\mathtt{Y} \notin \overline{\mathtt{X}}$. Since $\mathtt{Y} \notin \overline{\mathtt{X}}$ we have that $\sigma\mathtt{Y} = \mathtt{Y}$ and the result follows immediately, since $\mathtt{Y} \mathrel{<:} \mathtt{Y}$ by S-Refl.

Case CG-Fun: $V \vdash_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{R}}{\rightarrow}\mathtt{S} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{T}}{\rightarrow}\mathtt{U} \Rightarrow (\bigwedge \overline{C}) \wedge D$ and $V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \overline{\mathtt{T}} \mathrel{<:} \overline{\mathtt{R}} \Rightarrow \overline{C}$ and $V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{U} \Rightarrow D$, with $\overline{\mathtt{Y}} \cap V = \emptyset$ and $\overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset$. If we pick fresh $\overline{\mathtt{Z}}$ such that $\overline{\mathtt{Z}} \cap (V \cup \overline{\mathtt{X}}) = \emptyset$, it is easy to check that $V \cup \{\overline{\mathtt{Z}}\} \vdash_{\overline{\mathtt{X}}} [\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}} \mathrel{<:} [\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{R}} \Rightarrow \overline{C}$ and $V \cup \{\overline{\mathtt{Z}}\} \vdash_{\overline{\mathtt{X}}} [\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{S} \mathrel{<:} [\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{U} \Rightarrow D$. Now, since σ is a valid $\overline{\mathtt{X}}/(V \cup \{\overline{\mathtt{Z}}\})$-substitution, we can use the induction hypothesis to prove that $\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}} \mathrel{<:} \sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{R}}$ and $\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{S} \mathrel{<:} \sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{U}$. Using the subtyping rule for function types, we have $\mathtt{All}(\overline{\mathtt{Z}})\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{R}}{\rightarrow}[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{S} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Z}})\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}{\rightarrow}\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{U}$. The result now follows, since $\mathtt{All}(\overline{\mathtt{Z}})\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{R}}{\rightarrow}\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{S} = \sigma(\mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{R}}{\rightarrow}\mathtt{S})$ and $\mathtt{All}(\overline{\mathtt{Z}})\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}{\rightarrow}\sigma[\overline{\mathtt{Z}}/\overline{\mathtt{Y}}]\mathtt{U} = \sigma(\mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{T}}{\rightarrow}\mathtt{U})$. □

Proposition 3.4.2 (Completeness). *Let σ be an $\overline{\mathtt{X}}/V$-substitution with $\overline{\mathtt{X}} \cap V = \emptyset$, and let* S *and* T *be types such that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$. If $\sigma\mathtt{S} \mathrel{<:} \sigma\mathtt{T}$, then $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{T} \Rightarrow C$ for some C such that $\sigma \in C$.*

Proof. By induction on the structure of S and T.

Case: $\mathtt{S} = \mathtt{Y}$ where $\mathtt{Y} \in \overline{\mathtt{X}}$. We have $V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{T} \Rightarrow \{\mathtt{Bot} \mathrel{<:} \mathtt{Y} \mathrel{<:} \mathtt{R}\}$ where $\mathtt{T} \Downarrow^V \mathtt{R}$. Now, since σ is an $\overline{\mathtt{X}}/V$-substitution, we know that $FV(\sigma\mathtt{Y}) \cap \mathtt{V} = \emptyset$, and therefore, using Lemma 3.2.2, we have $\sigma\mathtt{Y} \mathrel{<:} \mathtt{R}$. This ensures that $\sigma \in \{\mathtt{Bot} \mathrel{<:} \mathtt{Y} \mathrel{<:} \mathtt{R}\}$ as required.

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

Case: $\mathtt{T} = \mathtt{Top}$. Immediate, since $V \vdash_{\overline{\mathtt{X}}} \mathtt{S} \mathrel{<:} \mathtt{Top} \Rightarrow \emptyset$ and $\sigma \in \emptyset$.

Case: $\mathtt{S} = \mathtt{Bot}$. Immediate, since $V \vdash_{\overline{\mathtt{X}}} \mathtt{Bot} \mathrel{<:} \mathtt{T} \Rightarrow \emptyset$ and $\sigma \in \emptyset$.

Case: $\mathtt{S} = \mathtt{Y}$ and $\mathtt{T} = \mathtt{Y}$ where $\mathtt{Y} \notin \overline{\mathtt{X}}$. Immediate, since $V \vdash_{\overline{\mathtt{X}}} \mathtt{Y} \mathrel{<:} \mathtt{Y} \Rightarrow \emptyset$ and $\sigma \in \emptyset$.

Case: $\mathtt{S} = \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{R}}{\rightarrow}\mathtt{R}$ and $\mathtt{T} = \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{U}}{\rightarrow}\mathtt{U}$. Since we identify type expressions up to alpha-conversion, we can pick $\overline{\mathtt{Y}}$ such 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 $\overline{\mathtt{X}}/(V \cup \{\overline{\mathtt{Y}}\})$-substitution and $\sigma\mathtt{S} = \mathtt{All}(\overline{\mathtt{Y}})\sigma\overline{\mathtt{R}}{\rightarrow}\sigma\mathtt{R}$ and $\sigma\mathtt{T} = \mathtt{All}(\overline{\mathtt{Y}})\sigma\overline{\mathtt{U}}{\rightarrow}\sigma\mathtt{U}$. Now, since $\sigma\mathtt{S} \mathrel{<:} \sigma\mathtt{T}$, it must be the case that $\sigma\overline{\mathtt{U}} \mathrel{<:} \sigma\overline{\mathtt{R}}$ and $\sigma\mathtt{R} \mathrel{<:} \sigma\mathtt{U}$. Using the induction hypothesis, $V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \overline{\mathtt{U}} \mathrel{<:} \overline{\mathtt{R}} \Rightarrow \overline{C}$ and $\sigma \in \overline{C}$. Similarly, $V \cup \{\overline{\mathtt{Y}}\} \vdash_{\overline{\mathtt{X}}} \mathtt{R} \mathrel{<:} \mathtt{U} \Rightarrow D$ and $\sigma \in D$. So, by CG-Fun, we have $V \vdash_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{R}}{\rightarrow}\mathtt{R} \mathrel{<:} \mathtt{All}(\overline{\mathtt{Y}})\overline{\mathtt{U}}{\rightarrow}\mathtt{U} \Rightarrow (\bigwedge \overline{C}) \wedge D$. Finally, by the fact that $(\bigwedge \overline{C}) \wedge D$ is a greatest lower bound, we have $\sigma \in (\bigwedge \overline{C}) \wedge D$, as required. □

3.5 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 type of the whole application as informative as possible. Depending on where the variables $\overline{\mathtt{X}}$ occur in R, this may involve choosing the smallest possible values for some variables and the largest for others. For example, if $\mathtt{R}$ is $\mathtt{X}\to\mathtt{Y}$ and we have generated the constraint set $\{\mathtt{S} \mathrel{<:} \mathtt{X} \mathrel{<:} \mathtt{T}, \mathtt{U} \mathrel{<:} \mathtt{Y} \mathrel{<:} \mathtt{V}\}$, then the smallest possible value for $\mathtt{R}$ is found by taking the substitution $[\mathtt{X} \mapsto \mathtt{T}, \mathtt{Y} \mapsto \mathtt{U}]$, which maximizes $\mathtt{X}$ and minimizes $\mathtt{Y}$.

It may also be the case that no substitution for the variables yields a minimal result type; for example, if $\mathtt{R}$ is $\mathtt{X}\to\mathtt{X}$ and we have the constraint set $\{\mathtt{Int} \mathrel{<:} \mathtt{X} \mathrel{<:} \mathtt{Top}\}$, 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 $\mathtt{X}$ affects the final result type.

(1) We say that $\mathtt{R}$ is *constant in* $\mathtt{X}$ when $[\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ for every $\mathtt{S}$ and $\mathtt{T}$.
(2) We say that $\mathtt{R}$ is *covariant in* $\mathtt{X}$ when $\Gamma \vdash [\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ iff $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.
(3) We say that $\mathtt{R}$ is *contravariant in* $\mathtt{X}$ when $\Gamma \vdash [\mathtt{T}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{S}/\mathtt{X}]\mathtt{R}$ iff $\Gamma \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$.
(4) We say that $\mathtt{R}$ is *invariant in* $\mathtt{X}$ when $\Gamma \vdash [\mathtt{S}/\mathtt{X}]\mathtt{R} \mathrel{<:} [\mathtt{T}/\mathtt{X}]\mathtt{R}$ iff $\mathtt{S} = \mathtt{T}$.

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

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. The *minimal substitution* $\sigma_{C\mathtt{R}}$ can be defined as follows:

```
For each $(\mathtt{S} \mathrel{<:} \mathtt{X}_i \mathrel{<:} \mathtt{T}) \in C$:
      if $\mathtt{R}$ is constant or covariant in $\mathtt{X}_i$
      then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{S}$
  else if $\mathtt{R}$ is contravariant in $\mathtt{X}_i$
      then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{T}$
  else if $\mathtt{R}$ is invariant in $\mathtt{X}_i$ and $\mathtt{S} = \mathtt{T}$
      then $\sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{S}$
  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 let σ be a $\overline{\mathtt{X}}/V$-substitution. We say that σ is *minimal* for C and $\mathtt{R}$ if $\sigma \in C$ and, for all $\overline{\mathtt{X}}/V$-substitutions σ' with $\sigma' \in C$, we have $\sigma\mathtt{R} \mathrel{<:} \sigma'\mathtt{R}$.

PROPOSITION 3.5.1.

(1) If the substitution $\sigma_{C\mathtt{R}}$ exists, then it is minimal 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 with $\sigma' \in C$. We must show that $\sigma_{C\mathtt{R}}\mathtt{R} \mathrel{<:} \sigma'\mathtt{R}$.
Let $n = |\overline{\mathtt{X}}|$. We can construct a sequence of substitutions $\sigma_0, \ldots, \sigma_n$ as follows:

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

Note that $\sigma_n = \sigma'$. We now argue that $\sigma_{i-1}\mathtt{R}$ <: $\sigma_i\mathtt{R}$ for each $i \geq 1$. Let $\mathtt{S}$ <: $\mathtt{X}_i$ <: $\mathtt{T}$ be the constraint associated with $\mathtt{X}_i$ in C.

—If $\mathtt{R}$ is constant or covariant in $\mathtt{X}_i$, then, by definition, $\sigma_{i-1}(\mathtt{X}_i) = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{S}$, and thus $\sigma_{i-1}(\mathtt{X}_i)$ <: $\sigma_i(\mathtt{X}_i)$. But this implies that $\sigma_{i-1}\mathtt{R}$ <: $\sigma_i\mathtt{R}$, by the definition of covariance.

—Similarly, if $\mathtt{R}$ is contravariant in $\mathtt{X}_i$, then $\sigma_{i-1}(\mathtt{X}_i) = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{T}$, and thus $\sigma_i(\mathtt{X}_i)$ <: $\sigma_{i-1}(\mathtt{X}_i)$, which implies that $\sigma_{i-1}\mathtt{R}$ <: $\sigma_i\mathtt{R}$, by the definition of contravariance.

—If $\mathtt{R}$ is invariant in $\mathtt{X}_i$, then $\sigma_{i-1}(\mathtt{X}_i) = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \mathtt{S}$, and we also know that $\mathtt{S} = \mathtt{T}$. But since $\mathtt{S}$ <: $\sigma_i(\mathtt{X}_i)$ <: $\mathtt{T}$, we have $\sigma_i(\mathtt{X}_i) = \mathtt{S}$, which, by the definition of invariance ($\sigma_{i-1}\mathtt{R} = \sigma_i\mathtt{R}$), yields $\sigma_{i-1}\mathtt{R}$ <: $\sigma_i\mathtt{R}$.

We have thus shown that $\sigma_{C\mathtt{R}}\mathtt{R} = \sigma_0\mathtt{R}$ <: $\sigma_1\mathtt{R}$ <: $\cdots$ <: $\sigma_n\mathtt{R} = \sigma'\mathtt{R}$, 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 $\mathtt{R}$. Since $\sigma_{C\mathtt{R}}$ is undefined, there is some $\mathtt{X}_i$ such that $\mathtt{R}$ is invariant in $\mathtt{X}_i$ but ($\mathtt{S}$ <: $\mathtt{X}_i$ <: $\mathtt{T}$) $\in C$ where $\mathtt{S} \neq \mathtt{T}$. Now, since $\sigma \in C$, we have that $\mathtt{S}$ <: $\sigma(\mathtt{X}_i)$ <: $\mathtt{T}$. Therefore, either the substitution $\sigma' = \sigma[\mathtt{X}_i \mapsto \mathtt{S}]$ or the substitution $\sigma' = \sigma[\mathtt{X}_i \mapsto \mathtt{T}]$ has the following properties: $\sigma' \in C$ and, by the definition of invariance, $\sigma\mathtt{R} \not<: \sigma'\mathtt{R}$. This contradicts our assumption that σ is minimal for C and $\mathtt{R}$. $\square$

COROLLARY 3.5.2. *The algorithmic rule*

$$\frac{\Gamma \vdash \mathtt{f} \in \mathtt{All}(\overline{\mathtt{X}})\overline{\mathtt{T}}\rightarrow\mathtt{R} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad |\overline{\mathtt{X}}| > 0 \qquad \emptyset \vdash_{\overline{\mathtt{X}}} \overline{\mathtt{S}} <: \overline{\mathtt{T}} \Rightarrow \overline{\mathtt{D}} \qquad C = \bigwedge \overline{D} \qquad \sigma = \sigma_{C\mathtt{R}}}{\Gamma \vdash \mathtt{f}(\overline{\mathtt{e}}) \in \sigma\mathtt{R} \Rightarrow \mathtt{f}[\sigma\overline{\mathtt{X}}](\overline{\mathtt{e}})} \quad \text{(App-InfAlg)}$$

is equivalent to the declarative rule given in Section 3.1.

4. BIDIRECTIONAL CHECKING

Our second type inference technique deals with the other kinds of undesirable type annotations identified in the introduction: annotations on bound variables in anonymous function abstractions and annotations on local variable bindings. We introduce a straightforward refinement of the internal language typing relation in which the typechecker operates two distinct modes: *synthesis* mode, where typing information is propagated upward from subexpressions, and *checking* mode, where information is propagated downward from enclosing expressions. Synthesis mode corresponds to the original typing rules of the internal language and is used when we do not know anything about the expected type of an expression (for top-level phrases,[5] function parts of application nodes, etc.). Checking mode is used when the surrounding context determines the type of the expression and we only need to check that it does have that type.

[5]In languages where modules have explicitly declared interfaces, it is possible that even top-level phrases could be processed in checking mode.

In an application node, the type of the function being applied determines the expected types of all the arguments. Suppose `f` has type `(Int→Int)→Int`, and consider the application `f(fun(x:Int)x)`. Because we know the type of `f`, we also know that the argument `fun(x:Int)x` must have type `Int→Int`, which determines the type annotation on the bound variable `x`—the type `Int` is the most specific (with respect to the subtype relation) that can validly be given to `x`. We therefore allow the annotation to be omitted, writing the whole application as `f(fun(x)x)`. During typechecking, `f`'s type is synthesized (by looking it up in the context), and then `fun(x)x` is processed in checking mode, with expected type `Int→Int`. (Note that we do not attempt to infer *type abstractions* automatically. A scheme for adding type binders as necessary in checking contexts would be a plausible extension to what we propose, but this seems less useful than inferring type annotations on ordinary abstractions. Also, employing such a scheme would mean that the binding sites of type variables would not always be lexically apparent.)

4.1 External Language Syntax

The external language for the system with bidirectional checking is identical to the one in the previous section, except that we allow an additional form of abstraction in which all value type annotations are omitted:

$$\mathtt{fun}[\overline{\mathtt{X}}](\overline{\mathtt{x}})\mathtt{e} \qquad \text{bare abstraction}$$

Note that we do not allow the type variable binders $[\overline{\mathtt{X}}]$ to be inferred. Also, for simplicity, abstractions have either full annotations or none (we could go further and allow some annotations to be included and others omitted on the same abstraction).

4.2 Type Inference

The bidirectional checking algorithm is formalized by splitting the type inference relation $\Gamma \vdash \mathtt{e} \in \mathtt{T} \Rightarrow \mathtt{e}'$ into two separate forms:

$$\begin{array}{ll} \Gamma \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{T} \Rightarrow \mathtt{e}' & \qquad \text{synthesis} \\ \Gamma \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{T} \Rightarrow \mathtt{e}' & \qquad \text{checking} \end{array}$$

The first form is read in the same way as the type inference relation in Section 3.1: "In context Γ, type annotations can be added to the external language term $\mathtt{e}$ to yield the internal language term $\mathtt{e}'$, which has type $\mathtt{T}$." The second can be read "In context Γ, type annotations can be added to $\mathtt{e}$ to yield $\mathtt{e}'$, which has a type smaller than $\mathtt{T}$."

In the rules that follow, we elide the "$\Rightarrow \mathtt{e}'$" part of both judgments, since it is always obvious how to calculate $\mathtt{e}'$. The rules themselves are mostly straightforward refinements of the typing rules for the internal language: the only real subtlety lies in determining when it is possible to switch from synthesis to checking mode. Each of the original typing rules is split into separate cases for synthesis and checking modes. For example, the synthesis rule for variables is identical to the rule in the internal language,

$$\Gamma \vdash \mathtt{x} \overset{\rightarrow}{\in} \Gamma(\mathtt{x}) \qquad \text{(S-Var)}$$

while the checking rule must perform an additional subtype check:

$$\frac{\Gamma \vdash \Gamma(\mathtt{x}) \mathrel{\mathtt{<:}} \mathtt{T}}{\Gamma \vdash \mathtt{x} \overset{\leftarrow}{\in} \mathtt{T}} \qquad \text{(C-Var)}$$

The synthesis rule for fully annotated abstractions is again identical to the internal language: we add the (explicitly given) annotations to the context and proceed in synthesis mode.

$$\frac{\Gamma, \overline{\mathtt{X}}, \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{T}}{\Gamma \vdash \mathtt{fun[}\overline{\mathtt{X}}\mathtt{]\,(}\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}\mathtt{)e} \overset{\rightarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{T}} \qquad \text{(S-Abs)}$$

There is no synthesis rule for unannotated function abstractions, since we cannot determine the missing type annotations from the local type information available. However, in a checking context, we can determine the appropriate annotations:

$$\frac{\Gamma, \overline{\mathtt{X}}, \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{T}}{\Gamma \vdash \mathtt{fun[}\overline{\mathtt{X}}\mathtt{]\,(}\overline{\mathtt{x}}\mathtt{)e} \overset{\leftarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{T}} \qquad \text{(C-Abs-Inf)}$$

If we encounter a fully annotated abstraction in a checking context, we check that the provided annotations are consistent with the type we are checking against:

$$\frac{\Gamma, \overline{\mathtt{X}} \vdash \overline{\mathtt{T}} \mathrel{\mathtt{<:}} \overline{\mathtt{S}} \qquad \Gamma, \overline{\mathtt{X}}, \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{R}}{\Gamma \vdash \mathtt{fun[}\overline{\mathtt{X}}\mathtt{]\,(}\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}}\mathtt{)e} \overset{\leftarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{T}}{\rightarrow}\mathtt{R}} \qquad \text{(C-Abs)}$$

The synthesis and checking rules for application nodes are again nearly identical: we synthesize the type of the function and then switch to checking mode for the arguments:

$$\frac{\Gamma \vdash \mathtt{f} \overset{\rightarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{R} \qquad \Gamma \vdash \overline{\mathtt{e}} \overset{\leftarrow}{\in} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}}{\Gamma \vdash \mathtt{f\,[}\overline{\mathtt{T}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)} \overset{\rightarrow}{\in} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{R}} \qquad \text{(S-App)}$$

In checking mode, we perform a final check that the actual result type is a subtype of the expected type.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \overset{\rightarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{S}}{\rightarrow}\mathtt{R}\\ \Gamma \vdash [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{R} \mathrel{\mathtt{<:}} \mathtt{U} \qquad \Gamma \vdash \overline{\mathtt{e}} \overset{\leftarrow}{\in} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\end{array}}{\Gamma \vdash \mathtt{f\,[}\overline{\mathtt{T}}\mathtt{]\,(}\overline{\mathtt{e}}\mathtt{)} \overset{\leftarrow}{\in} \mathtt{U}} \qquad \text{(C-App)}$$

To combine bidirectional checking and type argument synthesis, we also need synthesis and checking versions of the "bare application" rule from Section 3.1.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \overset{\rightarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{T}}{\rightarrow}\mathtt{R}\\ \Gamma \vdash \overline{\mathtt{e}} \overset{\rightarrow}{\in} \overline{\mathtt{S}} \qquad |\overline{\mathtt{X}}| > 0 \qquad \Gamma \vdash \overline{\mathtt{S}} \mathrel{\mathtt{<:}} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\\ \forall\overline{\mathtt{V}}.\ (\Gamma \vdash \overline{\mathtt{S}} \mathrel{\mathtt{<:}} [\overline{\mathtt{V}}/\overline{\mathtt{X}}]\overline{\mathtt{T}} \text{ implies } \Gamma \vdash [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{R} \mathrel{\mathtt{<:}} [\overline{\mathtt{V}}/\overline{\mathtt{X}}]\mathtt{R})\end{array}}{\Gamma \vdash \mathtt{f(}\overline{\mathtt{e}}\mathtt{)} \overset{\rightarrow}{\in} [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{R}} \qquad \text{(S-App-InfSpec)}$$

$$\frac{\Gamma \vdash \texttt{f} \overset{\rightarrow}{\in} \texttt{All}(\overline{\texttt{X}})\overline{\texttt{T}}\to\texttt{R} \qquad \Gamma \vdash \overline{\texttt{e}} \overset{\rightarrow}{\in} \overline{\texttt{S}} \\ |\overline{\texttt{X}}| > 0 \qquad \Gamma \vdash \overline{\texttt{S}} <: [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{T}} \qquad \Gamma \vdash [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{R} <: \texttt{V}}{\Gamma \vdash \texttt{f}(\overline{\texttt{e}}) \overset{\leftarrow}{\in} \texttt{V}} \qquad \text{(C-App-InfSpec)}$$

Note that the checking version of this rule is significantly more permissive than the synthesis version, since it allows any type arguments $\overline{\texttt{U}}$ which satisfy the appropriate constraints: there is no need to try to minimize the result type. This means that the checking rule will perform significantly better on polymorphic function types such as `All(X)()→(X→X)`, where the result type mentions a polymorphic variable in both positive and negative positions.

The expected type `Top` does not give any useful information in a checking context: when it appears, we simply revert to synthesis mode:

$$\frac{\Gamma \vdash \texttt{e} \overset{\rightarrow}{\in} \texttt{T}}{\Gamma \vdash \texttt{e} \overset{\leftarrow}{\in} \texttt{Top}} \qquad \text{(C-Top)}$$

Finally, we need checking and synthesis rules corresponding to the typing rule for `Bot`:

$$\frac{\Gamma \vdash \texttt{f} \overset{\rightarrow}{\in} \texttt{Bot} \qquad \Gamma \vdash \overline{\texttt{e}} \overset{\rightarrow}{\in} \overline{\texttt{S}}}{\Gamma \vdash \texttt{f}[\overline{\texttt{T}}](\overline{\texttt{e}}) \overset{\rightarrow}{\in} \texttt{Bot}} \qquad \text{(S-App-Bot)}$$

$$\frac{\Gamma \vdash \texttt{f} \overset{\rightarrow}{\in} \texttt{Bot} \qquad \Gamma \vdash \overline{\texttt{e}} \overset{\rightarrow}{\in} \overline{\texttt{S}}}{\Gamma \vdash \texttt{f}[\overline{\texttt{T}}](\overline{\texttt{e}}) \overset{\leftarrow}{\in} \texttt{R}} \qquad \text{(C-App-Bot)}$$

It is worth remarking that application expressions involving *both* type argument synthesis and anonymous function arguments (specifically, anonymous function arguments that are not thunks) are not handled well by our type inference rules, since we force the argument expressions to be synthesized. Fortunately, our measurements of ML code in Appendix A show that application expressions of this form only occur about once every 100 lines of code.

4.3 Local Variable Bindings

The above rules for typechecking function application embody a simple heuristic: synthesize the type of the function, and then use the resulting information to switch to checking mode for the argument expressions. This heuristic works well in contexts where the head of an application expression is a variable or another application expression, both of whose types can easily be synthesized.

One important case where our heuristic fails is in the encoding of let-expressions. The expression `let x = e in b` is normally encoded as `(fun(x)b) e`, which fails to typecheck, since the type of the function `fun(x)b` cannot be synthesized. A better approach would be to synthesize the type of `e` first, and then use that to determine the type of `x`. We could include a second typing rule for application expressions to do exactly this, synthesizing the argument expression types and then switching to checking mode for the function expression. However, this would introduce some nondeterminism in the typing of expressions and require backtracking in the typechecker implementation. A simpler solution would be to include let-expressions in the internal language, add the typechecking rules below, and leave the heuristic for typechecking application expressions unchanged.

$$\frac{\Gamma \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{S} \qquad \Gamma, \mathtt{x}\!:\!\mathtt{S} \vdash \mathtt{b} \overset{\rightarrow}{\in} \mathtt{T}}{\Gamma \vdash \mathtt{let\ x\ =\ e\ in\ b} \overset{\rightarrow}{\in} \mathtt{T}} \tag{S-Let}$$

$$\frac{\Gamma \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{S} \qquad \Gamma, \mathtt{x}\!:\!\mathtt{S} \vdash \mathtt{b} \overset{\leftarrow}{\in} \mathtt{T}}{\Gamma \vdash \mathtt{let\ x\ =\ e\ in\ b} \overset{\leftarrow}{\in} \mathtt{T}} \tag{C-Let}$$

4.4 Soundness and Completeness

Appropriate refinements of the soundness and partial completeness theorems of Section 3.1 can be shown to hold when bidirectional checking is added.

THEOREM 4.4.1 (SOUNDNESS).

(*1*) *If* $\Gamma \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{T} \Rightarrow \mathtt{e}'$, *then* $\mathtt{e}$ *is a partial erasure of* $\mathtt{e}'$ *and* $\Gamma \vdash \mathtt{e}' \in \mathtt{T}$.

(*2*) *If* $\Gamma \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{T} \Rightarrow \mathtt{e}'$, *then* $\mathtt{e}$ *is a partial erasure of* $\mathtt{e}'$ *and* $\Gamma \vdash \mathtt{e}' \mathrel{<:} \mathtt{T}$.

PROOF. By induction on derivations. □

THEOREM 4.4.2 (PARTIAL COMPLETENESS). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ *(i.e.,* $\mathtt{e}$ *is fully typed), then*

(*1*) $\Gamma \vdash \mathtt{e} \overset{\rightarrow}{\in} \mathtt{T} \Rightarrow \mathtt{e}$

(*2*) $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{U}$ *implies* $\Gamma \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{U} \Rightarrow \mathtt{e}$.

PROOF. By induction on derivations. □

(We might expect that the following stronger version of Theorem 4.4.2(2) would also hold:

If $\Gamma \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{T}$ and $\Gamma \vdash \mathtt{T} \mathrel{<:} \mathtt{U}$, then $\Gamma \vdash \mathtt{e} \overset{\leftarrow}{\in} \mathtt{U}$.

Unfortunately, this is not the case. For example, the checking rule for `fun` does not apply if the type constraint is `Top`.)

4.5 Calculating Type Arguments

The algorithmic version of the S-App-InfSpec rule is similar to the algorithmic rule App-InfAlg, which we presented in Section 3.5. The algorithmic version of the C-App-InfSpec rule is different, however, since we do not need to choose a substitution σ which minimizes the result type of the expression:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \overset{\rightarrow}{\in} \mathtt{All(}\overline{\mathtt{X}}\mathtt{)}\overline{\mathtt{T}}\mathtt{\rightarrow R} \qquad \Gamma \vdash \overline{\mathtt{e}} \overset{\rightarrow}{\in} \overline{\mathtt{S}} \qquad |\overline{\mathtt{X}}| > 0 \\ \emptyset \vdash_{\overline{\mathtt{X}}} \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}} \Rightarrow \overline{\mathtt{C}} \qquad \emptyset \vdash_{\overline{\mathtt{X}}} \mathtt{R} \mathrel{<:} \mathtt{V} \Rightarrow D \qquad \sigma \in \bigwedge \overline{C} \wedge D\end{array}}{\Gamma \vdash \mathtt{f(}\overline{\mathtt{e}}\mathtt{)} \overset{\leftarrow}{\in} \mathtt{V} \Rightarrow \mathtt{f[}\sigma\mathtt{X](}\overline{\mathtt{e}}\mathtt{)}} \tag{C-App-InfAlg}$$

That is, we calculate the set of constraints generated by the arguments just as in Section 3.5, and add in the constraints generated by comparing the result type R with the expected type V. If the combined constraints are satisfiable (i.e., if $\mathtt{S}_i$ <: $\mathtt{T}_i$ for each ($\mathtt{S}_i$ <: $\mathtt{X}_i$ <: $\mathtt{T}_i$) $\in \bigwedge \overline{C} \wedge D$), then we succeed; otherwise we fail.

5. LOCAL TYPE ARGUMENT SYNTHESIS WITH BOUNDED QUANTIFICATION

We now describe an optional extension to the local type argument synthesis technique described in Section 3 to include an internal language where bounded quantification is allowed (specifically, we treat Cardelli and Wegner's Kernel $F_{\leq}$—or "Kernel Fun" [Cardelli and Wegner 1985]—extended with Bot). All the properties presented above continue to hold for the extended system (including the combination with the bidirectional propagation technique described in Section 4), but the algorithms involved in generating constraint sets become somewhat more subtle, due principally to some surprising interactions between bounded quantifiers and the Bot type [Pierce 1997]. The treatment of Bot is not just "dual to Top," since bounds in $F_{\leq}$ are asymmetric: we have upper bounds for variables (such as X<:T) but no lower bounds (such as T<:X). In particular, the intuitive property that "a type variable has no subtypes except itself and Bot" fails to hold; for example, if the context contains X<:Bot, then we have X <: Y for any variable Y.

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 interdependent 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. It appears that this restriction could be relaxed if a more clever constraint solver were employed, but we do not see how to remove it completely.

5.1 Bounded Quantification

For our full explicitly typed internal language, we use Cardelli and Wegner's Kernel $F_{\leq}$ calculus [Cardelli and Wegner 1985] of subtyping and impredicative polymorphism, enriched with Bot. We only give definitions here; the metatheory of the system has been developed in detail elsewhere [Pierce 1997].

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

The only difference from the internal language defined in Section 2 is the addition of bounds to the quantifiers.

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

The rule for comparing function types in the subtyping relation is refined as follows:

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

Note that we use the original "Kernel" rule for comparing quantifiers [Cardelli and Wegner 1985], 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 [Curien and Ghelli 1992; Cardelli et al. 1994].[6] 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 $F_{\leq}$" [Ghelli 1990].

It is also important to note that some of the usual properties of presentations of Kernel $F_{\leq}$ without Bot do not hold here. For instance, $\Gamma \vdash \mathtt{S} \mathrel{\mathtt{<:}} \mathtt{T}$ and $\Gamma \vdash \mathtt{T} \mathrel{\mathtt{<:}} \mathtt{S}$ do not imply $\mathtt{S} = \mathtt{T}$ (consider, for example, $\mathtt{X}\mathtt{<:}\mathtt{Bot} \vdash \mathtt{X} \mathrel{\mathtt{<:}} \mathtt{Bot}$ and $\mathtt{X}\mathtt{<:}\mathtt{Bot} \vdash \mathtt{Bot} \mathrel{\mathtt{<:}} \mathtt{X}$). This fact is the result of the interaction between bounded quantification and Bot, and it substantially complicates the proofs of the properties in the remainder of this section. See Pierce [1997].

We write $\Gamma \vdash \mathtt{S} \Uparrow \mathtt{T}$ for the operation which calculates a *least non-variable supertype* T of a type S by repeated promotion of variables:

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

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

We write $\Gamma \vdash \mathtt{S} \wedge \mathtt{T} = \mathtt{M}$ for "M is the meet of S and T in context Γ" and $\Gamma \vdash \mathtt{S} \vee \mathtt{T} = \mathtt{J}$ for "J is the join of S and T in Γ." The definitions of these relations can be found in Pierce [1997, Section 3.3].

The rules for (multi-)abstractions and applications straightforwardly refine the original ones to deal with bounds:

$$\frac{\Gamma, \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}}, \overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}} \vdash \mathtt{e} \in \mathtt{T}}{\Gamma \vdash \mathtt{fun}[\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}}]\,(\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{S}})\mathtt{e} \in \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}})\overline{\mathtt{S}}\to\mathtt{T}} \qquad \text{(T-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \in \mathtt{F} \qquad \Gamma \vdash \mathtt{F} \Uparrow \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{B}})\overline{\mathtt{S}}\to\mathtt{R} \\ \Gamma \vdash \mathtt{T}_i \mathrel{\mathtt{<:}} [\mathtt{T}_1/\mathtt{X}_1 \ldots \mathtt{T}_{i-1}/\mathtt{X}_{i-1}]\mathtt{B}_i \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{U}} \mathrel{\mathtt{<:}} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\end{array}}{\Gamma \vdash \mathtt{f}\,[\overline{\mathtt{T}}]\,(\overline{\mathtt{e}}) \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{R}} \qquad \text{(T-App)}$$

[6]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 better overall.

$$\frac{\Gamma \vdash \mathtt{f} \in \mathtt{F} \qquad \Gamma \vdash \mathtt{F} \Uparrow \mathtt{Bot} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}}}{\Gamma \vdash \mathtt{f}\,[\overline{\mathtt{T}}]\,(\overline{\mathtt{e}}) \in \mathtt{Bot}} \qquad \text{(T-App-Bot)}$$

5.2 Type Inference (Specification)

The specification of type inference changes only a little from what we saw in Section 3.1.

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

(App-InfSpec)

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 $\Gamma \vdash \overline{\mathtt{A}} \mathtt{<:} \overline{\mathtt{S}}$ ensures that the arguments are subtypes of the required bounds $\overline{\mathtt{S}}$, while the condition $\Gamma \vdash \overline{\mathtt{U}} \mathtt{<:} [\overline{\mathtt{A}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$ 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 $[\overline{\mathtt{A}}/\overline{\mathtt{X}}]\mathtt{R}$.

5.3 Variable Elimination

In the constraint-generation algorithm, it will again 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. Of course, this promotion or demotion must now take place with respect to the more interesting subtyping relation of Kernel $F_{\leq}$—in particular, the promotion and demotion relations will be indexed by a context Γ.

The ability to eliminate variables in this way is a crucial reason for choosing the "Kernel" 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 [Ghelli and Pierce 1998].

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

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

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

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

$$\frac{\mathtt{X} \notin V}{\Gamma \vdash \mathtt{X} \Uparrow^V \mathtt{X}} \tag{VU-Var-2}$$

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

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

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

$$\Gamma \vdash \mathtt{Top} \Downarrow^V \mathtt{Top} \tag{VD-Top}$$

$$\Gamma \vdash \mathtt{Bot} \Downarrow^V \mathtt{Bot} \tag{VD-Bot}$$

$$\frac{\mathtt{X} \in V}{\Gamma \vdash \mathtt{X} \Downarrow^V \mathtt{Bot}} \tag{VD-Var-1}$$

$$\frac{\mathtt{X} \notin V}{\Gamma \vdash \mathtt{X} \Downarrow^V \mathtt{X}} \tag{VD-Var-2}$$

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

$$\frac{FV(\overline{\mathtt{A}}) \cap V \neq \emptyset}{\Gamma \vdash \mathtt{All(}\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{A}}\mathtt{)}\overline{\mathtt{S}}\mathtt{\rightarrow T} \Downarrow^V \mathtt{Bot}} \tag{VD-Fun-2}$$

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 Ghelli and Pierce [1998], but somewhat simpler because of the presence of `Bot` in our type system.)

Lemma 5.3.1 (Soundness of Variable Elimination).

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

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

Lemma 5.3.2 (Completeness of Variable Elimination).

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

Proof. See Pierce [1997]. □

5.4 Constraints

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

Formally, an $\overline{\mathtt{X}}/V$*-constraint* has one of the forms below, with the additional constraint that all the free variables of $\mathtt{S}$ and $\mathtt{T}$ are distinct from $V \cup \overline{\mathtt{X}}$.

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

A type $\mathtt{R}$ *satisfies* a constraint c, written $\Gamma \vdash \mathtt{R} \in c$, if

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

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

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

An $\overline{\mathtt{X}}/V$*-constraint set* C is a finite map from $\overline{\mathtt{X}}$ to $\overline{\mathtt{X}}/V$-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):

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

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 $C_1 \wedge \ldots \wedge C_m \wedge D_1 \wedge \ldots \wedge D_n$.

5.5 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 $\mathtt{S}$ and $\mathtt{T}$, calculates the minimal $\overline{\mathtt{X}}/V$-constraint set C guaranteeing that $\Gamma \vdash \mathtt{S} <: \mathtt{T}$.

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

5.6 Soundness and Completeness of Constraint Generation

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

LEMMA 5.6.1 (SOUNDNESS OF MATCHING CONSTRAINT GENERATION). *If* $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \mathtt{U}, C$ *and* $\sigma \in C$ *then* $\sigma\mathtt{S} = \sigma\mathtt{T} = \mathtt{U}$.

Proof. Straightforward induction. □

Lemma 5.6.2 (Completeness of Matching-Constraint Generation). *If σ is an $\overline{\mathtt{X}}/V$-substitution where $\overline{\mathtt{X}} \cap V = \emptyset$, and* S *and* T *are types such that either $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ or $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$, then $\sigma\mathtt{S} = \sigma\mathtt{T}$ implies that $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \sigma\mathtt{S}, C$ for some C such that $\Gamma \vdash \sigma \in C$.*

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 $\Gamma \vdash \sigma \in C$ and $\Gamma \vdash \sigma \in D$ implies $\Gamma \vdash \sigma \in C \wedge D$.

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 T occurs in the codomain of σ and σ is a $\overline{\mathtt{X}}/V$-substitution. We therefore have $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} \equiv \mathtt{T} \Rightarrow \mathtt{T}, \{\mathtt{Y} \mapsto [\mathtt{T}]\}$ and $\Gamma \vdash \sigma \in \{\mathtt{Y} \mapsto [\mathtt{T}]\}$ as required.

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

Proposition 5.6.3 (Soundness of Constraint Generation). *Suppose that $FV(\Gamma) \cap \overline{\mathtt{X}} = \emptyset$ and $dom(\Gamma) \cap \overline{\mathtt{X}} = \emptyset$. If $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{T} \Rightarrow C$ and $\Gamma \vdash \sigma \in C$, then $\Gamma \vdash \sigma\mathtt{S} <: \sigma\mathtt{T}$.*

Proof. By induction on the derivation of $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{T} \Rightarrow C$. Proceed by case analysis on the final rule used in the derivation.

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

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

Case: $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{Y} <: \mathtt{T} \Rightarrow C$ where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\Gamma \vdash \mathtt{T} \Downarrow^{V} \mathtt{R}$, $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$ and $C = \{\mathtt{Y} \mapsto [\mathtt{Bot}, \mathtt{R}]\}$. Since $\Gamma \vdash \sigma \in C$ we have $\Gamma \vdash \sigma\mathtt{Y} <: \mathtt{R}$. Since $FV(\mathtt{T}) \cap \overline{\mathtt{X}} = \emptyset$, we know that $\sigma\mathtt{T} = \mathtt{T}$; also, Lemma 5.3.1(2) tells us that $\Gamma \vdash \mathtt{R} <: \mathtt{T}$. We therefore have $\Gamma \vdash \sigma\mathtt{Y} <: \mathtt{R} <: \mathtt{T} = \sigma\mathtt{T}$, as required.

Case: $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{Y} \Rightarrow C$ where $\mathtt{Y} \in \overline{\mathtt{X}}$, $\Gamma \vdash \mathtt{S} \Uparrow^{V} \mathtt{R}$, $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$ and $C = \{\mathtt{Y} \mapsto [\mathtt{R}, \mathtt{Top}]\}$. Since $\Gamma \vdash \sigma \in C$ we have $\Gamma \vdash \mathtt{R} <: \sigma\mathtt{Y}$. Since $FV(\mathtt{S}) \cap \overline{\mathtt{X}} = \emptyset$, we know that $\sigma\mathtt{S} = \mathtt{S}$; also, Lemma 5.3.1(1) tells us that $\Gamma \vdash \mathtt{S} <: \mathtt{R}$. We therefore have $\Gamma \vdash \sigma\mathtt{S} = \mathtt{S} <: \mathtt{R} <: \sigma\mathtt{Y}$, as required.

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

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

Case: $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \mathtt{All}(\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{A}})\overline{\mathtt{R}}\mathtt{\to S} <: \mathtt{All}(\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{B}})\overline{\mathtt{T}}\mathtt{\to U} \Rightarrow D \wedge \overline{D} \wedge \overline{C}$ where $\Gamma \vdash^{V}_{\overline{\mathtt{X}}} \overline{\mathtt{A}} \equiv \overline{\mathtt{B}} \Rightarrow \overline{\mathtt{K}}, \overline{D}$ and $\Gamma, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \overline{\mathtt{T}} <: \overline{\mathtt{R}} \Rightarrow \overline{C}$ and $\Gamma, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{K}} \vdash^{V'}_{\overline{\mathtt{X}}} \mathtt{S} <: \mathtt{U} \Rightarrow D$ and $V' = V \cup \{\overline{\mathtt{Y}}\}$ and $\overline{\mathtt{Y}} \cap V = \emptyset$ and $\overline{\mathtt{Y}} \cap \overline{\mathtt{X}} = \emptyset$. We may assume (wlog) that the $\overline{\mathtt{Y}}$ are fresh variables—in particular, that $FV(\sigma) \cap \mathtt{Y} = \emptyset$ and that σ is a valid $\overline{\mathtt{X}}/V'$-substitution. Our assumption that $\Gamma \vdash \sigma \in D \wedge \overline{D} \wedge \overline{C}$, plus the fact that $FV(\overline{\mathtt{K}}) \cap \overline{\mathtt{X}} = \emptyset$, implies that we can use the induction hypothesis to prove $\Gamma, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{K}} \vdash \sigma\mathtt{S} <: \sigma\mathtt{U}$ and $\Gamma, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{K}} \vdash \sigma\overline{\mathtt{T}} <:$

$\sigma\overline{\texttt{R}}$. Moreover, Lemma 5.6.1 tells us that $\sigma\overline{\texttt{A}} = \sigma\overline{\texttt{B}} = \overline{\texttt{K}}$. By the subtyping rule for functions, we conclude $\Gamma \vdash \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{A}})\sigma\overline{\texttt{R}}\to\sigma\texttt{S} \texttt{ <: } \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}})\sigma\overline{\texttt{T}}\to\sigma\texttt{U}$. The result follows, since $\texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{A}})\sigma\overline{\texttt{R}}\to\sigma\texttt{S} = \sigma(\texttt{All}(\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{A}})\overline{\texttt{R}}\to\texttt{S})$ and $\texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}})\sigma\overline{\texttt{T}}\to\sigma\texttt{U} = \sigma(\texttt{All}(\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{B}})\overline{\texttt{T}}\to\texttt{U})$. ☐

Proposition 5.6.4 (Completeness of Constraint Generation). *Let σ be an $\overline{\texttt{X}}/V$-substitution with $\overline{\texttt{X}} \cap V = \emptyset$, and let* S *and* T *be types such that either $FV(\texttt{S}) \cap \overline{\texttt{X}} = \emptyset$ or $FV(\texttt{T}) \cap \overline{\texttt{X}} = \emptyset$. Let Γ be a context such that $\overline{\texttt{X}} \cap dom(\Gamma) = \emptyset$ and $FV(\Gamma) \cap \overline{\texttt{X}} = \emptyset$. If $\Gamma \vdash \sigma\texttt{S} \texttt{ <: } \sigma\texttt{T}$, then $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{S} \texttt{ <: } \texttt{T} \Rightarrow C$ and $\Gamma \vdash \sigma \in C$.*

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

Case: $\texttt{S} = \texttt{Y}$ where $\texttt{Y} \in \overline{\texttt{X}}$. We have $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{Y} \texttt{ <: } \texttt{T} \Rightarrow C$ where $C = \{\texttt{Y}\mapsto[\texttt{Bot},\texttt{R}]\}$ and $\texttt{T} \Downarrow^V \texttt{R}$. Now, since σ is a $\overline{\texttt{X}}/V$-substitution, we know that $FV(\sigma\texttt{Y}) \cap V = \emptyset$, and therefore, using Lemma 5.3.2, we have $\Gamma \vdash \sigma\texttt{Y} \texttt{ <: } \texttt{R}$. This ensures that $\Gamma \vdash \sigma \in \texttt{C}$, as required.

Case: $\texttt{T} = \texttt{Y}$ where $\texttt{Y} \in \overline{\texttt{X}}$. We have $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{S} \texttt{ <: } \texttt{Y} \Rightarrow C$ where $C = \{\texttt{Y}\mapsto[\texttt{R},\texttt{Top}]\}$ and $\texttt{S} \Uparrow^V \texttt{R}$. Now, since σ is a $\overline{\texttt{X}}/V$-substitution, we know that $FV(\sigma\texttt{Y}) \cap V = \emptyset$, and therefore, using Lemma 5.3.2, we have $\Gamma \vdash \texttt{R} \texttt{ <: } \sigma\texttt{Y}$. This ensures that $\Gamma \vdash \sigma \in \texttt{C}$, as required.

Case: $\texttt{T} = \texttt{Top}$. Immediate, since $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{S} \texttt{ <: } \texttt{Top} \Rightarrow \emptyset$ and $\Gamma \vdash \sigma \in \emptyset$.

Case: $\texttt{S} = \texttt{Bot}$. Immediate, since $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{Bot} \texttt{ <: } \texttt{T} \Rightarrow \emptyset$ and $\Gamma \vdash \sigma \in \emptyset$.

Case: $\texttt{S} = \texttt{Y}$ and $\texttt{T} = \texttt{Y}$ where $\texttt{Y} \notin \overline{\texttt{X}}$. Immediate, since $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{Y} \texttt{ <: } \texttt{Y} \Rightarrow \emptyset$ and $\sigma \in \emptyset$.

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

Case: $\Gamma \vdash \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{A}})\sigma\overline{\texttt{R}}\to\sigma\texttt{S} \texttt{ <: } \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}})\sigma\overline{\texttt{T}}\to\sigma\texttt{U}$, where $\sigma\overline{\texttt{A}} = \sigma\overline{\texttt{B}}$ and $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash \sigma\overline{\texttt{T}} \texttt{ <: } \sigma\overline{\texttt{R}}$ and $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash \sigma\texttt{S} \texttt{ <: } \sigma\texttt{U}$. Since we identify type expressions up to alpha-conversion, we may suppose (wlog) that the $\overline{\texttt{Y}}$ are chosen so that $\overline{\texttt{Y}} \cap V = \emptyset$, $\overline{\texttt{Y}} \cap \overline{\texttt{X}} = \emptyset$, and $FV(\sigma) \cap \overline{\texttt{Y}} = \emptyset$. If $V' = V \cup \overline{\texttt{Y}}$ then σ is a valid $\overline{\texttt{X}}/V'$-substitution and we can use the induction hypothesis to prove that $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash^{V'}_{\overline{\texttt{X}}} \overline{\texttt{T}} \texttt{ <: } \overline{\texttt{R}} \Rightarrow \overline{C}$ and $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash \sigma \in \overline{C}$, and similarly, $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash^{V'}_{\overline{\texttt{X}}} \texttt{S} \texttt{ <: } \texttt{U} \Rightarrow D$ and $\Gamma, \overline{\texttt{Y}}\texttt{<:}\sigma\overline{\texttt{B}} \vdash \sigma \in D$. Using Lemma 5.6.2, we have that $\Gamma \vdash^V_{\overline{\texttt{X}}} \overline{\texttt{A}} \equiv \overline{\texttt{B}} \Rightarrow \sigma\overline{\texttt{B}}, \overline{D}$ and $\Gamma \vdash \sigma \in \overline{D}$. So, by the constraint generation rule for function types, we have $\Gamma \vdash^V_{\overline{\texttt{X}}} \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{A}})\overline{\texttt{S}}\to\texttt{P} \texttt{ <: } \texttt{All}(\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{B}})\overline{\texttt{T}}\to\texttt{Q} \Rightarrow D \wedge \overline{D} \wedge \overline{C}$. Finally, by the fact that $D \wedge \overline{D} \wedge \overline{C}$ is a greatest lower bound, we have $\Gamma \vdash \sigma \in D \wedge \overline{D} \wedge \overline{C}$, as required. ☐

5.7 Calculating Type Arguments

As before, the first step in calculating the actual type arguments to an application begins by formalizing the ways in which maximizing or minimizing X affects the final result type. The main new element here is the case of rigid variables:

(1) We say that R is *constant in* X when $\Gamma \vdash [\texttt{S}/\texttt{X}]\texttt{R} \texttt{ <: } [\texttt{T}/\texttt{X}]\texttt{R}$ for every S and T.

(2) We say that R is *covariant in* X when $\Gamma \vdash [\texttt{S}/\texttt{X}]\texttt{R} \texttt{ <: } [\texttt{T}/\texttt{X}]\texttt{R}$ iff $\Gamma \vdash \texttt{S} \texttt{ <: } \texttt{T}$.

(3) We say that R is *contravariant in* X when $\Gamma \vdash [\texttt{T}/\texttt{X}]\texttt{R} \texttt{ <: } [\texttt{S}/\texttt{X}]\texttt{R}$ iff $\Gamma \vdash \texttt{S} \texttt{ <: } \texttt{T}$.

(4) We say that R is *invariant in* X when $\Gamma \vdash [\texttt{S}/\texttt{X}]\texttt{R} \texttt{ <: } [\texttt{T}/\texttt{X}]\texttt{R}$ iff both $\Gamma \vdash \texttt{S} \texttt{ <: } \texttt{T}$ and $\Gamma \vdash \texttt{T} \texttt{ <: } \texttt{S}$.

(5) We say that R is *rigid in* X when $\Gamma \vdash$ [S/X]R <: [T/X]R iff S = T.

It is easy to check whether R is constant, covariant, contravariant, invariant, or rigid in a given variable X by examining where X occurs in 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 X<:Bot, then X→X is equivalent, but not identical, to Bot→Bot: indeed its equivalence class has several members. On the other hand, Top is only equivalent to itself. Formally, we call a type variable a *bottom variable* (in Γ) if its upper bound is Bot or by another bottom variable. Now, let Γ be a context and S a type whose free variables are in *dom*(Γ). We say that S is *rigid under* Γ if

—S = Top;

—S = Bot and no variable in Γ is bounded by Bot;

—S = X and X is not a bottom variable;

—S = All($\overline{\texttt{X}}$<:$\overline{\texttt{A}}$)$\overline{\texttt{S}}$→T with each $\texttt{A}_i$ rigid under Γ, $\texttt{X}_1$<:$\texttt{A}_1, \ldots,\texttt{X}_{i-1}$<:$\texttt{A}_{i-1}$ and $\overline{\texttt{S}}$ and T rigid under Γ, $\overline{\texttt{X}}$<:$\overline{\texttt{A}}$;

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

LEMMA 5.7.1. *If* S *is rigid under* Γ*, then every type equivalent to* S *is syntactically equal to* S*—i.e.,* $\Gamma \vdash$ S <: T *and* $\Gamma \vdash$ T <: S *together imply that* S *and* T *are identical.*

PROOF. See Pierce [1997, Lemma 4.1.2]. □

COROLLARY 5.7.2. *If* c *is rigid under* Γ *and* $\Gamma \vdash$ S $\in c$ *and* $\Gamma \vdash$ T $\in c$*, then* S *and* T *are identical.*

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

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

else $\sigma_{C\mathtt{R}}$ is undefined.

We can again show:

PROPOSITION 5.7.3.

(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 suppose σ' is another substitution such that $\Gamma \vdash \sigma' \in C$. We must show that $\Gamma \vdash \sigma_{C\mathtt{R}}\mathtt{R} \mathrel{<:} \sigma'\mathtt{R}$.
Let $n = |\overline{\mathtt{X}}|$, and construct a sequence of substitutions $\sigma_0, \ldots, \sigma_n$ as follows:

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

Note that $\sigma_n = \sigma'$. We now argue that $\Gamma \vdash \sigma_{i-1}\mathtt{R} \mathrel{<:} \sigma_i\mathtt{R}$ for each $i \geq 1$.
—If $\mathtt{R}$ is constant or 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 $\Gamma \vdash \sigma_{i-1}(\mathtt{X}_i) \mathrel{<:} \sigma_i(\mathtt{X}_i)$. But this implies that $\Gamma \vdash \sigma_{i-1}\mathtt{R} \mathrel{<:} \sigma_i\mathtt{R}$, 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 $\Gamma \vdash \sigma_i(\mathtt{X}_i) \mathrel{<:} \sigma_{i-1}(\mathtt{X}_i)$, which implies that $\Gamma \vdash \sigma_{i-1}\mathtt{R} \mathrel{<:} \sigma_i\mathtt{R}$, by the definition of contravariance.
—If $\mathtt{R}$ is invariant in $\mathtt{X}_i$, then $\sigma_{i-1}\mathtt{X}_i = \sigma_{C\mathtt{R}}(\mathtt{X}_i) = \min(C(\mathtt{X}_i))$, and we also know (by the tightness of $C(\mathtt{X}_i)$) that $\Gamma \vdash \min(C(\mathtt{X}_i)) \mathrel{<:} \max(C(\mathtt{X}_i)) \mathrel{<:} \min(C(\mathtt{X}_i))$. But since $\Gamma \vdash \min(C(\mathtt{X}_i)) \mathrel{<:} \sigma_i(\mathtt{X}_i)) \mathrel{<:} \min(C(\mathtt{X}_i))$, we have by transitivity, $\Gamma \vdash \sigma_i(\mathtt{X}_i) \mathrel{<:} \sigma_{i-1}(\mathtt{X}_i) \mathrel{<:} \sigma_i(\mathtt{X}_i)$, which, by the definition of invariance, yields $\Gamma \vdash \sigma_{i-1}\mathtt{R} \mathrel{<:} \sigma_i\mathtt{R}$.
—Finally, if $\mathtt{R}$ is rigid in $\mathtt{X}_i$, then $\sigma_{i-1}(\mathtt{X}_i) = \sigma_i(\mathtt{X}_i)$, and so $\Gamma \vdash \sigma_{i-1}\mathtt{R} \mathrel{<:} \sigma_i\mathtt{R}$ by reflexivity of subtyping.
We have thus shown that $\Gamma \vdash \sigma_{C\mathtt{R}}\mathtt{R} = \sigma_0\mathtt{R} \mathrel{<:} \sigma_1\mathtt{R} \mathrel{<:} \cdots \mathrel{<:} \sigma_n\mathtt{R} = \sigma'\mathtt{R}$, 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 $\Gamma \vdash \mathtt{T} \in C(\mathtt{X}_i)$ but such that either $\Gamma \vdash \sigma(\mathtt{X}_i) \not\mathrel{<:} \mathtt{T}$ or $\Gamma \vdash \mathtt{T} \not\mathrel{<:} \sigma(\mathtt{X}_i)$. We can then construct a substitution $\sigma' = [\mathtt{X}_i \mapsto \mathtt{T}]$ such that $\Gamma \vdash \sigma' \in C$ and, since $\mathtt{X}_i$ is invariant in $\mathtt{R}$, such that $\Gamma \vdash \sigma\mathtt{R} \not\mathrel{<:} \sigma'\mathtt{R}$, 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 $\Gamma \vdash \mathtt{T} \in C(\mathtt{X}_i)$. We can then construct a substitution $\sigma' = \sigma[\mathtt{X}_i \mapsto \mathtt{T}]$ such that $\Gamma \vdash \sigma' \in C$ and, since $\mathtt{X}_i$ in rigid in $\mathtt{R}$, such that $\Gamma \vdash \sigma\mathtt{R} \not\mathrel{<:} \sigma'\mathtt{R}$, contradicting our assumption that σ is minimal for C and $\mathtt{R}$. $\square$

Corollary 5.7.4. *The algorithmic rule*

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{f} \uparrow \mathtt{All}(\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{S}})\overline{\mathtt{T}}\to\mathtt{R} \Rightarrow \mathtt{f}' \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{U}} \Rightarrow \overline{\mathtt{e}}' \qquad |\overline{\mathtt{X}}| > 0 \qquad \overline{\mathtt{X}} \cap FV(\overline{\mathtt{S}}) = \emptyset \\ \Gamma \vdash^{\emptyset}_{\overline{\mathtt{X}}} \overline{\mathtt{X}} \mathtt{<:}\ \overline{\mathtt{S}} \Rightarrow \overline{C} \qquad \Gamma \vdash^{\emptyset}_{\overline{\mathtt{X}}} \overline{\mathtt{U}} \mathtt{<:}\ \overline{\mathtt{T}} \Rightarrow \overline{D} \qquad E = (\overline{C} \wedge \overline{D}) \qquad \sigma = \sigma_{E\mathtt{R}}\end{array}}{\Gamma \vdash \mathtt{f}(\overline{\mathtt{e}}) \in \sigma\mathtt{R} \Rightarrow \mathtt{f}'[\sigma\overline{\mathtt{X}}](\overline{\mathtt{e}}')}$$

is equivalent to the declarative rule given in Section 5.2.

6. EXTENSIONS

We have experimented with these and similar type inference techniques in our compiler for the Pict language [Pierce and Turner 1997b]. Although these experiments do not yet cover the full language, they give some confidence that the methods do actually infer enough type annotations to be helpful. (Indeed, we converted around 10,000 lines of library code from a version of Pict incorporating Cardelli's greedy algorithm to one using a variant of the techniques presented here in a few hours.) Moreover, they provide an indication of how well these techniques scale to languages with more features than the tiny core calculus presented here. In general, our experience has been quite encouraging: it has usually been quite easy to see how to extend the definitions here to the larger syntax and richer type system found in Pict.

However, one important set of issues remains incompletely resolved. A significant difference between Pict's type system and the variants of $F_{\leq}$ studied here and in Pierce and Turner [1997a] is that Pict includes type operators—formally, it is based on the higher-order extension $F^{\omega}_{\leq}$ [Cardelli 1990; Cardelli and Longo 1991; Pierce and Turner 1994; Pierce and Steffen 1994; Hofmann and Pierce 1995; Compagnoni 1994]. Our type argument synthesis technique needs to know whether type operators are covariant, contravariant, or invariant in the subtype relation; in the case of $F^{\omega}_{\leq}$, this requires that we distinguish covariant, contravariant, and invariant user-defined type operators. The necessary extension of $F^{\omega}_{\leq}$ with *polarized type operators* is significantly more complex than the form in which $F^{\omega}_{\leq}$ is usually studied [Compagnoni 1994; Pierce and Steffen 1994], and its metatheoretic properties are a matter of current investigation [Steffen 1998]. We are experimenting with strategies for simplifying the system and have achieved some promising preliminary results.

Another important avenue for further investigation is the possibility of combining these type inference techniques with overloading. There is reason to hope that the integration can be accomplished smoothly, at least for limited forms of overloading, since we have insisted that each typable term should have a unique manifest type. (This property plays a crucial role in the formulation of simple overloading systems like Java's: the type of an argument to an overloaded operator must be uniquely determined before overloading resolution.)

7. RELATED WORK

There have been a number of proposals for partial type inference schemes treating just impredicative polymorphism (without subtyping). One line of work has been explored by Pfenning [1988b; Pfenning [1993], following earlier work of Boehm

[Boehm 1985; 1989]. Interestingly, the key algorithm here comes from a proof of *un*decidability of a certain style of partial type inference, where occurrences of type application must be marked but the type argument itself need not be supplied, and where all other type annotations may be omitted. Boehm showed that this form of type inference was just as hard as higher-order unification, hence undecidable. Conversely, Huet's earlier work on efficient semi-algorithms for higher-order unification [Huet 1975] led directly to a useful semi-algorithm for partial type inference [Pfenning 1988b]. Later improvements in this line of development have included using a more refined algorithm for higher-order constraint solving [Dowek et al. 1996], eliminating the troublesome possibilities of nontermination or generation of non-unique solutions. Experience with related algorithms in languages such as LEAP [Pfenning and Lee 1991], Elf [Pfenning 1989], and FX [O'Toole and Gifford 1989] has shown them to be quite well behaved in practice.

A different approach to partial type inference (still without subtyping) was initiated by Läufer and Odersky [1994], sparked by Perry's observation that first-class existential types can be added to ML by integrating them with the `datatype` mechanism [Perry 1990]. In essence, `datatype` constructors and destructors can be regarded as explicit type annotations, marking where values must be injected into and projected from disjoint union types, where recursive types must be folded and unfolded, and (when existentials are added) where packing and unpacking must occur. This idea was extended to include first-class (impredicative) universal quantifiers by Rémy [1994]. Other, more recent, proposals by Odersky and Läufer [1996] and Garrigue and Rémy [1997] conservatively extend ML-style type inference by allowing programmers to explicitly annotate function arguments with types, which may (unlike the annotations that can be inferred automatically) contain embedded universal quantifiers, thus partly bridging the gap between ML and System F. This family of approaches to type inference has the advantage of relative simplicity and clean integration with the existing Hindley/Milner polymorphism of ML.

We know of only one partial type inference scheme that works in the presence of both impredicative polymorphism and subtyping: Cardelli's "greedy type inference algorithm" for $F_{\leq}$ [Cardelli 1993]. (Similar algorithms have also been used in proof-checkers for dependent type theories, such as NuPrl [Howe 1988] and Lego [Pollack 1990].) The idea here is that any type annotation may be omitted by the programmer: a fresh unification variable α will be generated for each one by the parser. During typechecking, the subtype-checking algorithm may be asked to check whether some type `S` is a subtype `T`, where both `S` and `T` may contain unification variables. Subtype-checking proceeds as usual until a subgoal of the form α `<: T` or `T <:` α is encountered, at which point α is instantiated to `T`, thus satisfying the immediate constraint in the simplest possible way. Of course, setting α to `T` may not be the best possible choice, and this may cause later subtype-checks for types involving α to fail when a different choice would have allowed them to succeed; but, again, practical experience with this algorithm in Cardelli's implementation and in an early version of the Pict language [Pierce and Turner 1997b] shows that the algorithm's greedy choice is correct in nearly all cases.

Unfortunately, there are some situations in which the greedy algorithm is almost guaranteed to guess wrong. For example, if `f` has type `(S,T)`$\to$`Int` and `T <: S` then the expression `fun(x) f(x,x)` will fail to typecheck: the greedy algorithm first assigns x the indeterminate type α; after checking the first argument to f it concludes that α must equal S. But then the second argument check fails, since we should have given x type T. In such cases, the algorithm's behavior can be quite puzzling to the programmer, yielding mysterious errors far from the point where a suboptimal instantiation is made.

Also, we should note that Cardelli's greedy algorithm lacks *monotonicity*: it is not the case that adding some type annotations will always improve the chances that the algorithm will be able to find the rest. Formally, there is a fully typed term e, a partial erasure e′ of e, and a further erasure e″ of e′, such that e and e″ pass the type inference algorithm, while e′ does not. (For the greedy algorithm, this failure was first noticed by Dilip Sequeira.) While this kind of behavior has never been observed in practice, we would be happier to see it excluded in principle. It is currently an open question whether our proposed type inference algorithm behaves well in this respect.

The difficulties with the greedy algorithm can be traced to the fact that there is no way of giving a robust explanation of its behavior without describing the typing, subtyping, and unification algorithms in complete detail, since the instantiations that they perform are highly sensitive to the precise order in which constraints are encountered during checking. This means that the language definition, to be complete, must describe the internal structure of the compiler in quite a bit of detail. Our goal in this article has been to develop partial type inference methods that share the good behavior in common cases of the greedy algorithm, but that are much more straightforward to explain to programmers.

Although we focus here on the combination of subtyping and polymorphism, it is worth remarking that there are other ways of achieving a synthesis of object-oriented and ML-style programming, not necessarily involving subtyping. Currently, the most successful design is Objective Caml, an object-oriented dialect of ML now in use in a number of software projects worldwide [Rémy and Vouillon 1997]. A crucial design choice in Objective Caml is the use of *row-variable polymorphism* [Wand 1987; 1988; Rémy 1989; Wand 1994] instead of *subsumption* for the typing of objects and classes. In Objective Caml, an object with a large interface cannot simply be regarded as an object with a smaller interface; however, it is straightforward to write functions that manipulate both kinds of objects by "quantifying over the difference" between their interfaces. The type inference algorithm aids the programmer by performing this kind of generalization wherever possible.

8. DISCUSSION

We have identified a promising class of *local* type inference methods and studied two representatives in detail. To evaluate the contributions of these two particular methods, let us review the requirements stated in the introduction:

(1) *To make fine-grained polymorphism tolerable, type arguments in applications of polymorphic functions must usually be inferred. However, it is acceptable to require annotations on the bound variables of top-level function definitions (since these usually provide useful documentation) and local function definitions (since these are relatively rare).*

We have seen that our local type argument synthesis method is complete for a certain class of situations—those in which either (1) some choice of values for the omitted type parameters yields a (unique) minimal result type for the whole application, or (2) the application itself appears in a checking context. How common these situations will be in practice is an empirical question that is difficult to address until some good-sized programs have been written in languages supporting ML-style programming with subtyping. However, we can get some feeling for the coverage of our type inference techniques by examining a few typical examples.

To make the examples more familiar, suppose that our core language has been extended with list types `List(T)` (the type of lists whose elements have type `T`) and reference types `Ref(T)` (the type of mutable storage cells containing elements of `T`). The `List` type constructor may soundly be taken to be covariant—i.e., we have `List(S) <: List(T)` whenever `S <: T`—while `Ref` must be invariant—i.e., we have `Ref(S) <: Ref(T)` only when `S = T`. These types come with the following built-in constants and functions:

```
nil     ∈  List(Bot)
cons    ∈  All(X) (X, List(X))→List(X)
map     ∈  All(X,Y) (List(X), X→Y)→List(Y)
newref  ∈  All(X) X→Ref(X)
deref   ∈  All(X) Ref(X)→X
update  ∈  All(X) Ref(X)→X→Unit
```

Assuming we are also given integers and arithmetic operators and that the variables `l` and `r` have types `List(Int)` and `Ref(Int)`, we have the following simple examples:

`cons(1, cons(2, cons(3, nil)))`	succeeds by (1)
`map(l, fun(x:Int)x+1)`	succeeds by (1)
`newref(2)`	fails
`update(newref(2), 3)`	fails
`(fun(s:Ref(Int)) update(s,0)) (newref(2))`	succeeds by (2)
`update(r, 3)`	succeeds by (1)
`deref(r)`	succeeds by (1)

Our proposal does require annotations on all bound variables of function definitions. For *top-level* function definitions, we regard these annotations as beneficial anyway. For local function definitions, we would prefer to have these annotations inferred, since these type annotations are often "obvious" to the programmer and so do not provide significant value as documentation, but our measurements indicate that local function definitions are not too common in any case.

It is also worth noting that annotations on recursively defined functions (if our language had them) could never be inferred using our scheme. While this is a limitation, it does have some benefits. For example, polymorphic recursion is automatically supported. In Haskell, polymorphic recursion is allowed if top-level binders are annotated, which effectively represents a step in the direction of our methods.

(2) *To make higher-order programming convenient, it is helpful, though not absolutely necessary, to infer the types of parameters to anonymous function definitions.*
Bidirectional typechecking allows type annotations on anonymous abstractions to be omitted whenever they appear in checking contexts—for example, when they are used as arguments to functions. For example, if the function `f` has the type `(Int→Int)→Int`, we can write

```
    f (fun(x)x+3)
```

instead of:

```
    f (fun(x:Int)x+3)
```

The one exception is when the application expression in which an anonymous abstraction appears as argument omits some expected type arguments. For example, we cannot infer types in:

```
    map(l, fun(x)x+2)
```

Instead, we must provide either the type argument

```
    map[Int](l, fun(x)x+2)
```

or else the argument type of the anonymous abstraction:

```
    map(l, fun(x:Int)x+2)
```

(3) *To support a mostly functional style (where the manipulation of pure data structures leads to many local variable bindings), local bindings should not normally require explicit annotations.*
We are able to calculate the types of locally bound values as long as they can be *synthesized.* This means that almost all local bindings except functions will have their types inferred. Local function bindings must have their bound variables fully annotated with types.

One weakness of our proposal is the relative complexity of extending local type argument synthesis to handle bounded quantification. On the positive side, the strengths of our inference techniques include their simple descriptions, their predictability, their robustness in the face of extensions to the internal language, and their tendency to report errors close to the point where more type annotations are required (or where an actual error is present in the program).

More generally, restricting attention to local methods imposes several important design constraints on both the internal language and on possible type inference algorithms:

—Unification or matching can be used only during the processing of single nodes in the syntax tree: types involving unification variables are never added to the context, passed down as checking constraints, or returned as the results of type synthesis.

—Polymorphic applications must be fully *un*curried in order to obtain the benefits of type inference. Curried applications can still be used, but they are second-class in this respect. (This point is a corollary of the first.)

—Expressions in the internal language must have unique manifest types that can be calculated easily by the programmer, in order for the behavior of partial type inference to be predictable.

—The type system of the internal language must be sufficiently complete and regular to permit "best annotations" to be inferred. In the system studied here, this means in particular that the minimal type `Bot` must be provided, with some attendant increase in the complexity of the internal language (particularly when the system is extended to include bounded quantification). Similarly, type operators like `List` must be made covariant in the subtype relation in order to allow inference of type arguments to `nil` and `cons`.

APPENDIX

A. MEASUREMENTS

This appendix presents in more detail our measurements of the uses of type inference in ML programs, as a rough guide to the frequency of undesirable type annotations of various sorts that would arise if we adopted an ML programming style in a language with no type inference at all.

It is helpful to distinguish between two kinds of type annotations. One kind we call *reasonable*, the other *silly*—the difference being that reasonable type annotations have some value as documentation, while silly annotations do not. Obviously, opinions will vary on precisely which annotations belong in each category, but many cases are fairly clear. For example, type annotations on parameters to top-level function definitions are arguably reasonable, since (except for very short functions) they are not normally obvious and writing them explicitly helps make code more readable (moreover, they are *checked* documentation and can never be out of date).[7] On the other hand, it is hard to imagine why anyone would want to write or read either of the occurrences of `Int` in `cons[Int](3,nil[Int])`. They are both silly.

We are interested in the kinds and frequencies of type annotations that will typically arise if we adopt the programming style encouraged by ML in an explicitly typed language. The three characteristic features of this style—fine-grained polymorphism, higher-order programming, and heavy use of data constructors and destructors instead of mutable state—each lead to an increase in the number of type annotations; moreover, many of these annotations are silly.

The use of *fine-grained polymorphism*, in which individual functions (rather than whole modules, as in C++, Pizza, or GJ) are parameterized on type arguments, leads to type annotations whenever polymorphic functions are defined or used—e.g., the three occurrences of `[X]` in:

```
let cons-twice =
   fun[X] (v:X, l:List(X))
     cons[X](v, cons[X](v, nil[X]))
```

The abstraction on `X` is arguably reasonable (indeed, in many languages, it actually has behavioral significance), but the `[X]` arguments to `nil` and `cons` are silly.

[7]In fact, even in ML, many top-level definitions are given explicit type declarations in module signatures.

A *higher-order* programming style, in which small anonymous functions are passed as arguments to other functions, leads to an increase in the total number of functions. Moreover (unlike top-level function definitions), the types of the parameters to these functions are mostly obvious from context. For example, suppose `fold-range` is a function of type `(((Int,Int)→Int),Int,Int,Int)→Int`; we might use it in an expression like

```
fold-range(
    fun(x:Int, y:Int) x+y,
    0, 1, 10)
```

to calculate the sum of the numbers from `1` to `10`. The two occurrences of `Int` are silly annotations, since they act only to lengthen the expression and obscure its behavior; it would be clearer to write:

```
fold-range(
    fun(x,y) x+y,
    0, 1, 10)
```

A *mostly functional* (or, in the extreme, *purely functional*) style, which favors the construction of new data values rather than in-place mutation of existing ones, leads to an increase in the number of local variable bindings compared to an imperative style. An imperative program with one local declaration

```
let x : Int = 0;
x := x + 1;
x := x * 2;
x := x - 3;
return x;
```

can become a functional program with four:

```
let x : Int = 0 in
let y : Int = x + 1 in
let z : Int = y * 2 in
let r : Int = z - 3 in
r
```

Again, the type annotations on these binders are all silly. (The annotation on the single binder in the imperative version is also silly, but this matters less if such declarations are relatively rare.)

We chose the Objective Caml compiler as our experimental tool, because the front end is quite easy to understand and modify.[8] We gathered raw data by instrumenting the compiler to produce a trace showing where the generalization and instantiation operations were being used during typechecking, where function definitions were encountered, and so on for each of the quantities we were interested in measuring. Each program was then compiled in the usual way, and a small script was used to tabulate and summarize the resulting traces.[9]

[8]Although Objective Caml supports object-oriented idioms in addition to a “pure ML style,” this facility is relatively new and is not used heavily in the code we measured.

[9]The raw traces from which the tables in this section were generated are available on-line through `http://www.cis.upenn.edu/~bcpierce/lti-stats`.

We measured several publicly available Objective Caml programs, amounting to about 160,000 lines of code plus about 30,000 lines in interface files.

	lines (.ml)	lines (.mli)
CamlTk	10080	4596
Coq	69571	9054
Ensemble	27747	6842
MMM	15645	2967
OCaml Libs	8521	4746
OCaml Progs	27069	3872

Camltk, written at Inria-Roquencourt, is a collection of mainly stub functions providing an interface to the Tk toolkit. Coq, the largest single program we measured, is a theorem prover, also from INRIA. Ensemble is a toolkit for group communication in distributed systems, built at Cornell. MMM is a web browser, from INRIA. Finally, we included the Objective Caml system itself, dividing it into libraries (the `stdlib` and `otherlibs` subdirectories of the distribution) and the compiler itself (plus debugger, etc.). We included comments in the line counts, since we are interested in the impact of the presence or absence of type annotations on the full text that programmers actually read and write.

The discussion above identified three ways in which silly type annotations arise from features of the programming style promoted by ML. The first was fine-grained polymorphism, which encourages the use of large numbers of polymorphic functions. To estimate the impact of this feature in practice, we counted the frequency of instantiations of polymorphic variables and constructors[10] performed during type-checking: each instantiation would correspond to one or more type arguments in an explicitly typed language. We counted separately the instantiations arising from comparison functions (`=`, `<`, etc.), which are polymorphic in Objective Caml but could well be monomorphic in other languages.

	variable instantiation	constructor instantiation	comparison instantiation
CamlTk	13.1	28.9	1.2
Coq	38.8	32.1	2.1
Ensemble	19.1	16.0	2.4
MMM	14.8	20.4	1.4
OCaml Libs	13.7	9.5	5.2
OCaml Progs	16.9	9.8	1.9

To highlight the impact of including or eliding type annotations associated with various language features, we express our results (here and in the tables that follow) as numbers of occurrences per hundred lines of code. For example, in CamlTk, an instantiation occurs, on average, in 13.1% of the lines of code. Assuming 50 lines per screenful of text, this means that we might expect, on average, to see six or seven per displayed page.

The frequencies of constructor instances in this table should be taken with a grain of salt, since they include instantiations occurring during typechecking of patterns,

[10]The constructor instance count also includes instances arising from polymorphic record labels.

which can probably be avoided in many cases. The high frequency of instantiation in Coq is a consequence of its extensive use of Objective Caml's built-in stream syntax.

Another source of silly type annotations is type annotations on bound variables of anonymous functions. To gauge the importance of this effect, we counted the frequency of anonymous function definitions in each of the sample programs. (For simplicity, we did not count the number of arguments to each function definition or the sizes of the type annotations that would have been required if they had been written explicitly.)

	anonymous functions
CamlTk	2.9
Coq	12.4
Ensemble	2.4
MMM	2.8
OCaml Libs	0.7
OCaml Progs	3.1

We see that the usage of anonymous functions varies according to programming style: the Objective Caml libraries use almost none, preferring direct recursive definitions, while application programs tend to make reasonably frequent use of higher-order functions like `map` and `fold`. Coq uses a relatively high number of anonymous functions—a consequence, again, of its extensive use of Objective Caml's stream syntax, which is translated internally into calls to the lazy stream library involving large numbers of thunks.

Two final sources of silly type annotations are variable bindings and local function definitions. Since all definitions, including function definitions, are translated internally into `let`-bindings, we divide this count into three: local function definitions (probably silly), top-level function definitions (probably reasonable), and `let`-bindings of other kinds (probably silly).

	local functions	top-level functions	other let-bindings
CamlTk	0.5	7.5	8.7
Coq	1.5	7.0	10.5
Ensemble	2.8	4.2	9.6
MMM	1.0	3.8	8.8
OCaml Libs	0.6	8.7	7.9
OCaml Progs	0.5	3.9	6.9

Let-bindings are fairly frequent, as might be expected. Local functions are much less frequent than top-level definitions—but, especially in Ensemble, not as rare as we might have had hoped (given that we do not infer these). It is also interesting to note, in passing, that library code—CamlTk and the Objective Caml libraries—tends to define smaller functions than most of the application code.

As we noted for anonymous functions, these numbers give only a rough measure of the "cost" of adding type annotations, since more than one type annotation may be required for each `let`-binding. Also, small changes in programming style can make a large difference in the number and size of required annotations. For example, changing a Caml function definition from the form

```
let f = function <pat> → <exp> | ...
```

to the form

```
let f x:T = match x with <pat> → <exp> | ...
```

eliminates the need for explicit annotations in all of the patterns.

We also gathered some measurements to help evaluate the limitations of our proposed inference techniques. In particular, there are some situations where either, but not both, can be used. This occurs when a polymorphic function or constructor is applied to an argument list that includes an anonymous abstraction. We break the measurements of these "hard applications" into two categories—one where some function argument is really hard and the easier case where the function argument is actually a thunk (whose parameter is either `_` or `()`, and which can therefore easily be synthesized).

	"hard" function args	"hard" thunk args
CamlTk	1.7	0.0
Coq	1.9	9.7
Ensemble	1.1	0.1
MMM	0.8	0.0
OCaml Libs	0.4	0.0
OCaml Progs	1.1	0.0

Finally, we found it interesting to measure how often the generalization operation was used during typechecking: these would each correspond to one or more type abstractions in an explicitly typed language. As above, we distinguish between polymorphic top-level definitions and local definitions of polymorphic functions.

	top-level polymorphism	local polymorphism
CamlTk	0.4	0.1
Coq	2.9	0.5
Ensemble	2.2	0.8
MMM	0.4	0.1
OCaml Libs	2.0	0.1
OCaml Progs	0.6	0.0

There is actually considerable variation in the frequency of type generalization in the different styles of code represented in this table—much more than the variation in numbers of instantiations. Also, the frequency of generalization seems to have little correlation with the distinction between library and application code.

ACKNOWLEDGMENTS

This article synthesizes insights from conversations with more people than we can list—probably almost everyone we know—but a few contributions were particularly direct: John Reynolds first acquainted us [BCP] with the idea of bidirectional typechecking around 1988, while early discussions with Luca Cardelli helped plant the ideas about type argument synthesis that eventually developed into the proposal in Section 3 in this article. Work with Dilip Sequeira on refinements of Cardelli's greedy inference algorithm greatly improved our understanding of its good and bad properties. Scott Smith, Frank Pfenning, Konstantin Läufer, and Didier Remy gave us useful background on related work. Discussions with Robert Harper, John Reppy, Karl Crary, and Stephanie Weirich and careful comments from Haruo Hosoya and the POPL and TOPLAS referees significantly improved the final version.

REFERENCES

ADITYA, S. AND NIKHIL, R. S. 1991. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*. Number 523 in Lecture Notes in Computer Science. Springer-Verlag. Also available as MIT CSG Memo 329, June 1991.

AIKEN, A. AND WIMMERS, E. L. 1993. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*. ACM press, 31–41.

BOEHM, H.-J. 1985. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*. IEEE, 339–345.

BOEHM, H.-J. 1989. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*. Portland, OR, 192–206.

BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, C. Chambers, Ed. ACM SIGPLAN Notices volume 33 number 10. Vancouver, BC, 183–200.

CARDELLI, L. 1990. Notes about $\mathrm{F}^{\omega}_{<:}$. Unpublished manuscript.

CARDELLI, L. 1991. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds. Springer-Verlag. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

CARDELLI, L. 1993. An implementation of $F_{<:}$. Research report 97, DEC Systems Research Center. Feb.

CARDELLI, L. AND LONGO, G. 1991. A semantic basis for Quest. *Journal of Functional Programming 1,* 4 (Oct.), 417–458. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

CARDELLI, L., MARTINI, S., MITCHELL, J. C., AND SCEDROV, A. 1994. An extension of system F with subtyping. *Information and Computation 109,* 1–2, 4–56. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17,* 4 (Dec.), 471–522.

COMPAGNONI, A. B. 1994. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*. Kazimierz, Poland. Springer *Lecture Notes in Computer Science* 933, June 1995. Also available as University of Edinburgh, LFCS technical report ECS-LFCS-94-281, titled "Subtyping in $F^{\omega}_{\wedge}$ is decidable".

CURIEN, P.-L. AND GHELLI, G. 1992. Coherence of subsumption: Minimum typing and type-checking in $\mathrm{F}_{\leq}$. *Mathematical Structures in Computer Science 2*, 55–91. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

DOWEK, G., HARDIN, T., KIRCHNER, C., AND PFENNING, F. 1996. Unification via explicit substitutions: The case of higher-order patterns. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, M. Maher, Ed. MIT Press, Bonn, Germany, 259–273.

EIFRIG, J., SMITH, S., AND TRIFONOV, V. 1995. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Pro-*

gramming Semantics Conference. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier.

Flanagan, C. and Felleisen, M. 1997. Componential set-based analysis. *ACM SIGPLAN Notices 32,* 5 (May), 235–248.

Garrigue, J. and Rémy, D. 1997. Extending ML with semi-explicit polymorphism. In *International Symposium on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, M. Abadi and T. Ito, Eds. Springer-Verlag, 20–46.

Ghelli, G. 1990. Proof theoretic studies about a minimal type system integrating inclusion and parametric polymorphism. Ph.D. thesis, Università di Pisa. Technical report TD–6/90, Dipartimento di Informatica, Università di Pisa.

Ghelli, G. and Pierce, B. 1998. Bounded existentials and minimal typing. *Theoretical Computer Science 193*, 75–96.

Girard, J.-Y. 1972. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

Hofmann, M. and Pierce, B. 1995. A unifying type-theoretic framework for objects. *Journal of Functional Programming 5,* 4 (Oct.), 593–635. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

Hosoya, H. and Pierce, B. C. 1999. How good is local type inference? Tech. Rep. MS-CIS-99-17, University of Pennsylvania. June. Available from the authors.

Howe, D. 1988. Automating reasoning in an implementation of constructive type theory. Ph.D. thesis, Cornell University.

Huet, G. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science 1*, 27–57.

Jagannathan, S. and Wright, A. 1995. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium.* LNCS, vol. 983. Springer-Verlag, 207–224.

Läufer, K. and Odersky, M. 1994. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16,* 5 (Sept.), 1411–1430. An earlier version appeared in the Proceedings of the ACM SIGPLAN Workshop on ML and its Applications, 1992, under the title "An Extension of ML with First-Class Abstract Types".

Miller, D. 1992. Unification under a mixed prefix. *Journal of Symbolic Computation 14,* 4 (Oct.), 321–358.

Odersky, M. and Läufer, K. 1996. Putting type annotations to work. In *Conference Record of POPL '96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, St. Petersburg, Florida, 54–67.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Principles of Programming Languages (POPL).*

O'Toole, J. W. and Gifford, D. K. 1989. Type reconstruction with first-class polymorphic values. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon.* ACM Press, 207–217.

Perry, N. 1990. The implementation of practical functional programming languages. Ph.D. thesis, Imperial College.

Pfenning, F. 1988a. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.* ACM Press, Snowbird, Utah, 153–163.

Pfenning, F. 1988b. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah.* ACM Press, 153–163. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

PFENNING, F. 1989. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE Computer Society Press, Pacific Grove, California, 313–322.

PFENNING, F. 1993. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae 19,* 1,2, 185–199. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

PFENNING, F. AND LEE, P. 1991. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science 89,* 1 (21 Oct.), 137–159. Preliminary version in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.

PIERCE, B. AND STEFFEN, M. 1994. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET).* Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

PIERCE, B. C. 1997. Bounded quantification with bottom. Tech. Rep. 492, Computer Science Department, Indiana University.

PIERCE, B. C. AND TURNER, D. N. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4,* 2 (Apr.), 207–247. Preliminary version in Principles of Programming Languages (POPL), 1993.

PIERCE, B. C. AND TURNER, D. N. 1997a. Local type argument synthesis with bounded quantification. Tech. Rep. 495, Computer Science Department, Indiana University. Jan.

PIERCE, B. C. AND TURNER, D. N. 1997b. Pict: A programming language based on the pi-calculus. Tech. Rep. CSCI 476, Computer Science Department, Indiana University. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

POLLACK, R. 1990. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes.

POTTIER, F. 1997. Simplifying subtyping constraints. In *Proceedings of the International Conference on Functional Programming (ICFP).*

RÉMY, D. 1989. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin.* ACM, 242–249. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

RÉMY, D. 1994. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, M. Hagiya and J. C. Mitchell, Eds. Springer-Verlag, Sendai, Japan, 321–346.

RÉMY, D. AND VOUILLON, J. 1997. Objective ML: A simple object-oriented extension of ML. In *Conference Record of POPL '97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, Paris, France, 40–53. Full version to appear in *Theory and Practice of Object Systems, 1998.*

REYNOLDS, J. 1974. Towards a theory of type structure. In *Proc. Colloque sur la Programmation.* Springer-Verlag LNCS 19, New York, 408–425.

STEFFEN, M. 1998. Polarized higher-order subtyping. Ph.D. thesis, Universität Erlangen-Nürnberg. Forthcoming.

SULZMANN, M., ODERSKY, M., AND WEHR, M. 1997. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4).* Full version in *Theory and Practice of Object Systems, 1998.*

TRIFONOV, V. AND SMITH, S. 1996. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium.* LNCS, vol. 1145. Springer Verlag, 349–365.

WAND, M. 1987. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science.* Ithaca, NY.

WAND, M. 1988. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science.*

Wand, M. 1994. Type inference for objects with instance variables and inheritance. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, C. A. Gunter and J. C. Mitchell, Eds. The MIT Press, 97–120.

Wells, J. B. 1994. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*. 176–185.

Received November 1997; revised February 1999; accepted July 1999