# Local Type Inference

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

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

## Abstract

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.

## 1 Introduction

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


POPL 98 San Diego CA USA


Our goal is to develop simple, well-behaved type inference techniques for new language designs in the style of Quest [Car91], Pizza [OW97], or ML2000—designs supporting both object-oriented programming idioms and the characteristic coding styles of languages such as ML and Haskell. It has recently become fashionable to refer to these languages as HOT ("higher-order, typed"). By extension, we can speak of a *HOT programming style*—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.

In particular, we are concerned with languages whose type-theoretic core combines subtyping and impredicative polymorphism in the style of System F [Gir72, Rey74]. 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 [Wel94], 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 recently been reported [AW93, EST95, JW95, TS96, SOW97, FF97, Pot97, etc.], but practical checkers based on these results have yet to see widespread use.)

### 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 the HOT programming style (polymorphic instantiation, anonymous function abstractions, and local variable bindings) 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 HOT 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.
- Local variable bindings occur once every 12 lines, but, in all but one of the programs we measured, local definitions of functions only occur once every 66 lines.

These observations give a fairly clear indication of the properties that a type inference scheme should have in order to support a HOT 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 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.

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.

### Local Type Inference

In this paper, 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 remainder of the paper is organized as follows. In the next section, we define a fully typed internal language. Sections 3 and 4 develop the techniques of local synthesis of type arguments and bidirectional checking in detail. Section 5 sketches some possible extensions. Section 6 surveys related work. Section 7 offers evaluation and concluding remarks. Details of our measurements of ML programs appear in Appendix A. Proofs are omitted in this extended abstract; they can be found in an accompanying technical report [PT97b].

## 2 Internal Form

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 form*.
2. An *external form* in which some type annotations are made optional or omitted entirely. This is the language that the programmer actually uses. (In some 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 form and the internal one. (The terms *type inference*, *type reconstruction*, and *type synthesis* have all been used for this relation, with slightly different meanings. 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 form 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 leads to unportable programs.

Our internal language—the target for the type inference methods described in Sections 3 and 4—is based on System $F_{\leq}$, Cardelli and Wegner's core calculus of subtyping and impredicative polymorphism. We consider here 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 here can be extended to deal with bounded quantifiers (see Section 5 and [PT97a]), but the simple language presented here 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 $F_{\leq}$ [CW85] 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. 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 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 |
| | | $\texttt{All}(\overline{\texttt{X}})\overline{\texttt{T}}\texttt{→T}$ | function type |
| e | ::= | x | variable |
| | | $\texttt{fun}[\overline{\texttt{X}}](\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}})\texttt{e}$ | abstraction |
| | | $\texttt{e}[\overline{\texttt{T}}](\overline{\texttt{e}})$ | application |
| Γ | ::= | • | empty context |
| | | Γ, x:T | variable binding |
| | | Γ, X | type variable binding |

We use the meta-variables `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 Γ(x) for the type of x in Γ.

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

Types, terms, contexts, and judgements that differ only in the names of bound variables are regarded as identical. Binders in contexts are assumed to have distinct names; when a new binding is added to a context, we assume that it has been renamed so as to maintain this invariant. The rules for scoping of bound variables are as usual (in $\texttt{All}(\overline{\texttt{X}})\overline{\texttt{S}}\texttt{→T}$, the variables $\overline{\texttt{X}}$ are in scope in $\overline{\texttt{S}}$ and `T`). $FV(\texttt{T})$, the set of type variables free in `T`, is defined is the usual way.

### 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 $\overline{\texttt{S}} \texttt{<:} \overline{\texttt{T}}$ to mean "$|\overline{\texttt{S}}| = |\overline{\texttt{T}}|$ and $\texttt{S}_i \texttt{<:} \texttt{T}_i$ for all $1 \leq i \leq |\texttt{S}|$."

$$\texttt{X} \texttt{<:} \texttt{X} \qquad \text{(S-Refl)}$$

$$\texttt{T} \texttt{<:} \texttt{Top} \qquad \text{(S-Top)}$$

$$\texttt{Bot} \texttt{<:} \texttt{T} \qquad \text{(S-Bot)}$$

$$\frac{\overline{\texttt{T}} \texttt{<:} \overline{\texttt{R}} \qquad \texttt{S} \texttt{<:} \texttt{U}}{\texttt{All}(\overline{\texttt{X}})\overline{\texttt{R}}\texttt{→S} \texttt{<:} \texttt{All}(\overline{\texttt{X}})\overline{\texttt{T}}\texttt{→U}} \qquad \text{(S-Fun)}$$

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 (cf. [PT97b]). We use the notation $\texttt{S} \vee \texttt{T}$ to denote the least upper bound of `S` and `T`, and $\texttt{S} \wedge \texttt{T}$ for the greatest lower bound of `S` and `T`. The definitions of these relations can be found in [PT97b].

### 2.3 Explicit Typing Rules

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

The typing rule for variables is standard.

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

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

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

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.

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

($\Gamma \vdash \texttt{e} \texttt{<:} \texttt{T}$ here is an abbreviation for "$\Gamma \vdash \texttt{e} \in \texttt{S}$ and $\texttt{S} \texttt{<:} \texttt{T}$.")

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

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

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

**2.3.1 Theorem [Uniqueness of manifest types]:** If $\Gamma \vdash e \in S$ and $\Gamma \vdash e \in T$, then $S = T$.

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 Appendix A) 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 HOT programming. As an 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 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 x ∈ 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. All we need to do is to define a three-place *type inference* relation:

$$\Gamma \vdash e \in T \Rightarrow e'$$

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 $\Gamma \vdash e \in T$, the type inference relation contains an analogous rule with conclusion $\Gamma \vdash e \in T \Rightarrow e'$, 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:

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

The condition $|\overline{X}| > 0$ says that type argument synthesis is only required in the case where the function f is polymorphic but there are no explicit type arguments. (For simplicity, we don't synthesize type arguments in the case where an application node provides some, but not all, of its required type arguments explicitly. This would be easy to do, but does not seem very useful in practice.)

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

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 $e'$ if e can be obtained from $e'$ by erasing some type annotations (i.e., deleting type arguments from one or more applications).

### 3.1.1 Theorem [Soundness]:

If $\Gamma \vdash e \in T \Rightarrow e'$, then e is a partial erasure of $e'$ and $\Gamma \vdash e' \in T$.

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

It should be emphasized that the rule given above (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 the type until we reach a supertype in which these variables do not occur. Formally, we write $S \Uparrow^V T$ for the relation "T is the least supertype of S such that $FV(T) \cap V = \emptyset$." Fortunately, such a type can always be found. For example, suppose $V = \{X\}$; then $(X, Int) \to X \Uparrow^V (Bot, Int) \to Top$.

The variable-elimination relation can be computed as follows:

$$\texttt{Top} \Uparrow^V \texttt{Top} \qquad \text{(VU-Top)}$$

$$\texttt{Bot} \Uparrow^V \texttt{Bot} \qquad \text{(VU-Bot)}$$

$$\frac{X \in V}{X \Uparrow^V \texttt{Top}} \qquad \text{(VU-Var-1)}$$

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

$$\frac{\overline{S} \Downarrow^V \overline{U} \qquad T \Uparrow^V V}{\texttt{All}(\overline{X})\overline{S} \to T \Uparrow^V \texttt{All}(\overline{X})\overline{U} \to V} \qquad \text{(VU-Fun)}$$

The relation $S \Downarrow^V T$ in the last clause is defined analogously:

$$\texttt{Top} \Downarrow^V \texttt{Top} \qquad \text{(VD-Top)}$$

$$\texttt{Bot} \Downarrow^V \texttt{Bot} \qquad \text{(VD-Bot)}$$

$$\frac{X \in V}{X \Downarrow^V \texttt{Bot}} \qquad \text{(VD-Var-1)}$$

$$\frac{X \notin V}{X \Downarrow^V X} \qquad \text{(VD-Var-2)}$$

$$\frac{\overline{S} \Uparrow^V \overline{U} \qquad T \Downarrow^V V}{\texttt{All}(\overline{X})\overline{S} \to T \Downarrow^V \texttt{All}(\overline{X})\overline{U} \to V} \qquad \text{(VD-Fun)}$$

*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 [GP97], but somewhat simpler because of the presence of Bot in our type system.

## 3.3 Constraint Generation

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

$$\{S_i <: X_i <: T_i \mid (FV(S_i) \cup FV(T_i)) \cap V = \emptyset\}.$$

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

$$\left\{ S \vee U <: X_i <: T \wedge V \;\middle|\; \begin{array}{l} S <: X_i <: T \in C \text{ and} \\ U <: X_i <: V \in D \end{array} \right\}$$

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

Our constraint generation rules have the form

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

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

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

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

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

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

$$\frac{Y \in \overline{X} \qquad S \Downarrow^{V} T}{V \vdash_{\overline{X}} Y <: S \Rightarrow \{\mathtt{Bot} <: Y <: T\}}$$

$$\frac{Y \in \overline{X} \qquad S \Uparrow^{V} T}{V \vdash_{\overline{X}} S <: Y \Rightarrow \{T <: Y <: \mathtt{Top}\}}$$

$$V \vdash_{\overline{X}} Y <: Y \Rightarrow \emptyset$$

$$\frac{V \cup \{\overline{Y}\} \vdash_{\overline{X}} \overline{T} <: \overline{R} \Rightarrow \overline{C} \qquad V \cup \{\overline{Y}\} \vdash_{\overline{X}} S <: U \Rightarrow D}{V \vdash_{\overline{X}} \mathtt{All}(\overline{Y})\overline{R} \to S <: \mathtt{All}(\overline{Y})\overline{T} \to U \Rightarrow \bigwedge \overline{C} \wedge D}$$

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

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

Suppose $\sigma$ is an $\overline{X}/V$-substitution and $\overline{X} \cap V = \emptyset$. We say that $\sigma$ *satisfies* an $\overline{X}/V$-constraint set $C$, written $\sigma \in C$, iff $S_i <: \sigma(X_i) <: T_i$ for each $(S_i <: X_i <: T_i) \in C$. 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 $S_i <: T_i$ for each $(S_i <: X_i <: T_i) \in C$.

If $C$ and $D$ are two $\overline{X}/V$-constraint sets such that $\sigma \in C$ implies $\sigma \in D$ for all $\sigma$, 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).

**3.4.1 Proposition [Soundness]:** Suppose that either $FV(S) \cap \overline{X} = \emptyset$ or $FV(T) \cap \overline{X} = \emptyset$. If $V \vdash_{\overline{X}} S <: T \Rightarrow C$ and $\sigma \in C$, then $\sigma S <: \sigma T$.

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

### 3.5 Calculating Type Arguments

Having generated a set of constraints for the missing type parameters $\overline{X}$, the final job of the local constraint solver is to choose values for $\overline{X}$ that make the result type of the whole application as informative as possible. Depending on where the variables $\overline{X}$ occur in R, this may involve choosing the smallest possible values for some variables and the largest for others. For example, if R is $X \to Y$ and we have generated the constraint set $\{S <: X <: T, U <: Y <: V\}$, then the smallest possible value for R is found by maximizing X and minimizing Y—i.e., by taking the substitution $[X \mapsto T, Y \mapsto U]$.

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

1. We say that R is *covariant in* X if $\Gamma \vdash [S/X]R <: [T/X]R$ whenever $\Gamma \vdash S <: T$.
2. We say that R is *contravariant in* X if $\Gamma \vdash [T/X]R <: [S/X]R$ whenever $\Gamma \vdash S <: T$.
3. We say that R is *invariant in* X if $\Gamma \vdash [S/X]R <: [T/X]R$ only when $S = T$.

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

We can now show how to choose values for the variables $\overline{X}$ that will minimize R (or determine that this is not possible). Let $C$ be a satisfiable $\overline{X}/V$-constraint set. The *minimal substitution* $\sigma_{CR}$ can be defined as follows:

For each $(S <: X_i <: T) \in C$:
  if R is covariant in $X_i$
  then $\sigma_{CR}(X_i) = S$
else if R is contravariant in $X_i$
  then $\sigma_{CR}(X_i) = T$
else if R is invariant in $X_i$ and $S = T$
  then $\sigma_{CR}(X_i) = S$
else $\sigma_{CR}$ is undefined.

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

**3.5.1 Proposition:**

1. If the substitution $\sigma_{CR}$ exists, then it is a minimal substitution for $C$ and R.
2. If $\sigma_{CR}$ is undefined, then $C$ and R have no minimal substitution.

**3.5.2 Corollary:** The algorithmic rule

$$\frac{\Gamma \vdash f \in \mathtt{All}(\overline{X})\overline{T} \to R \qquad \Gamma \vdash \overline{e} \in \overline{S} \qquad |\overline{X}| > 0 \qquad \emptyset \vdash_{\overline{X}} \overline{S} <: \overline{T} \Rightarrow \overline{C} \qquad \sigma \in \bigwedge \overline{C} \Downarrow R}{\Gamma \vdash f(\overline{e}) \in \sigma R \Rightarrow f[\sigma\overline{X}](\overline{e})}$$

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, 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; for example, in an application node, the type of the function being applied determines the expected types of all the arguments.

For example, 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`.

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, e.g., [AN91]); it is surprisingly powerful when used by itself as a local type inference method. Specific technical contributions of this paper 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.

### 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:

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

Note that we do not allow the type variable binders $[\overline{X}]$ 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 $\Gamma \vdash e \in T \Rightarrow e'$ into two separate forms:

$$\Gamma \vdash e \vec{\in} T \Rightarrow e' \qquad \text{synthesis}$$
$$\Gamma \vdash e \overleftarrow{\in} T \Rightarrow e' \qquad \text{checking}$$

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

In the rules that follow, we elide the "$\Rightarrow e'$" part of both judgements, since it is always obvious how to calculate $e'$. 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,

$$\Gamma \vdash x \vec{\in} \Gamma(x)$$

while the checking rule must perform an additional subtype check.

$$\frac{\Gamma \vdash \Gamma(x) <: T}{\Gamma \vdash x \overleftarrow{\in} T}$$

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.

$$\frac{\Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash e \vec{\in} T}{\Gamma \vdash \texttt{fun}[\overline{X}]\,(\overline{x}{:}\overline{S})e \vec{\in} \texttt{All}(\overline{X})\overline{S}{\to}T}$$

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:

$$\frac{\Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash e \overleftarrow{\in} T}{\Gamma \vdash \texttt{fun}[\overline{X}]\,(\overline{x})e \overleftarrow{\in} \texttt{All}(\overline{X})\overline{S}{\to}T}$$

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:

$$\frac{\Gamma, \overline{X} \vdash \overline{T} <: \overline{S} \qquad \Gamma, \overline{X}, \overline{x}{:}\overline{S} \vdash e \overleftarrow{\in} R}{\Gamma \vdash \texttt{fun}[\overline{X}]\,(\overline{x}{:}\overline{S})e \overleftarrow{\in} \texttt{All}(\overline{X})\overline{T}{\to}R}$$

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:

$$\frac{\Gamma \vdash f \vec{\in} \texttt{All}(\overline{X})\overline{S}{\to}R \qquad \Gamma \vdash \overline{e} \overleftarrow{\in} [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash f[\overline{T}]\,(\overline{e}) \vec{\in} [\overline{T}/\overline{X}]R}$$

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

$$\frac{\Gamma \vdash f \vec{\in} \texttt{All}(\overline{X})\overline{S}\to R \qquad \Gamma \vdash [\overline{T}/\overline{X}]R <: U \qquad \Gamma \vdash \overline{e} \overleftarrow{\in} [\overline{T}/\overline{X}]\overline{S}}{\Gamma \vdash f[\overline{T}](\overline{e}) \overleftarrow{\in} U}$$

Note that the above rules for function application embody a simple heuristic: always synthesize the type of the function, and then use the resulting information to switch to checking mode for the argument expressions. The reason this heuristic works well is that the head of an application expression is almost always a variable or another application expression, and we can easily synthesize the types of both kinds of expression. It is possible, of course, to come up with examples where it would be beneficial to synthesize the argument types first and then use the resulting information to avoid type annotations in the function part of an application expression. For example, we could infer that x has type Int in the expression (fun(x)e)(3), since the argument 3 has type Int. Unfortunately, this refinement does not help infer the types of polymorphic functions. For example, we cannot uniquely determine the type of x in the expression (fun[X](x)e)[Int](3). (Note also that adding a second typing rule for application expressions would introduce some non-determinism in the typing of expressions and require some backtracking in the typechecker implementation.)

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

$$\frac{\begin{array}{c}\Gamma \vdash f \vec{\in} \texttt{All}(\overline{X})\overline{T}\to R \\ \Gamma \vdash \overline{e} \vec{\in} \overline{S} \qquad |\overline{X}| > 0 \qquad \Gamma \vdash \overline{S} <: [\overline{U}/\overline{X}]\overline{T} \\ \forall \overline{V}.\ (\Gamma \vdash \overline{S} <: [\overline{V}/\overline{X}]\overline{T} \text{ implies } \Gamma \vdash [\overline{U}/\overline{X}]R <: [\overline{V}/\overline{X}]R)\end{array}}{\Gamma \vdash f(\overline{e}) \vec{\in} [\overline{U}/\overline{X}]R}$$

$$\frac{\Gamma \vdash f \vec{\in} \texttt{All}(\overline{X})\overline{T}\to R \qquad \Gamma \vdash \overline{e} \vec{\in} \overline{S} \qquad |\overline{X}| > 0 \qquad \Gamma \vdash \overline{S} <: [\overline{U}/\overline{X}]\overline{T} \qquad \Gamma \vdash [\overline{U}/\overline{X}]R <: V}{\Gamma \vdash f(\overline{e}) \overleftarrow{\in} V}$$

Note that the checking version this rule is significantly more permissive than the synthesis version, since it allows any type arguments $\overline{U}$ 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:

$$\frac{\Gamma \vdash e \vec{\in} T}{\Gamma \vdash e \overleftarrow{\in} \texttt{Top}}$$

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

$$\frac{\Gamma \vdash f \vec{\in} \texttt{Bot} \qquad \Gamma \vdash \overline{e} \vec{\in} \overline{S}}{\Gamma \vdash f[\overline{T}](\overline{e}) \vec{\in} \texttt{Bot}}$$

$$\frac{\Gamma \vdash f \vec{\in} \texttt{Bot} \qquad \Gamma \vdash \overline{e} \vec{\in} \overline{S}}{\Gamma \vdash f[\overline{T}](\overline{e}) \overleftarrow{\in} R}$$

Its 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 per 100 lines of code.)

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

## 5 Extensions

In [PT97a], we show how to extend the local type argument synthesis technique described in Section 3 to an internal language where bounded quantification is allowed (specifically, we treat Cardelli and Wegner's Kernel Fun [CW85] extended with Bot). All the properties presented here continue to hold for the extended system (including the combination with the bidirectional propagation technique), 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 [Pie97]. (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.) Moreover, we impose a slight restriction on the types of polymorphic functions for which argument types can be inferred, disallowing dependencies between type arguments in a single application. 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.

We have experimented with these and similar type inference techniques in our compiler for the Pict language [PT97c]. 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 [PT97a] is that Pict includes type operators—formally, it is based on the higher-order extension $F^{\omega}_{\leq}$. Our type argument synthesis technique depends on the fact that type operators like List are covariant in the subtype relation; in the case of $F^{\omega}_{\leq}$, we must also recognize when user-defined type operators are co- or contravariant. 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 [PS94, Com94], and its meta-theoretic properties are a matter of current investigation [Ste97]. 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, since we have insisted that each typable term should have a unique manifest type.

## 6 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 [Pfe88, Pfe93], following earlier work of Boehm [Boe85, Boe89]. 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 [Hue75] led directly to a useful semi-algorithm for partial type inference [Pfe88]. Later improvements in this line of development have included using a more refined algorithm for higher-order constraint solving [DHKP96], eliminating the troublesome possibilities of nontermination or generation of non-unique solutions. Experience with related algorithms in languages such as LEAP [PL91], Elf [Pfe89], and FX [JG89] 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 [LO94], sparked by Perry's observation that first-class existential types can be added to ML by integrating them with the `datatype` mechanism [Per90]. 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 Remy [Rém94]. Other, more recent, proposals by Odersky and Läufer [OL96] and Rémy and Garrigue [GR97] 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 $F_{\leq}$ [Car93]. The idea here is that any type annotation may be omitted by the programmer: a fresh unification variable $\alpha$ 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 $\alpha$ `<:` `T` or `T` `<:` $\alpha$ is encountered, at which point $\alpha$ is instantiated to `T`, thus satisfying the immediate constraint in the simplest possible way. Of course, setting $\alpha$ to `T` may not be the best possible choice, and this may cause later subtype-checks for types involving $\alpha$ 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 [PT97c] 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)→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 $\alpha$; after checking the first argument to `f` it concludes that $\alpha$ 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 but $e'$ 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 any better 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 paper 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 HOT programming styles. The most successful design to date is Objective Caml, an object-oriented dialect of ML now in use in a number of software projects worldwide [RV97]. A crucial design choice in Objective Caml is the use of *row-variable polymor-*

*phism* [Wan87, Wan88, Rém89, Wan94] 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.

## 7 Conclusions

We have identified a promising class of *local* type inference methods and studied two representatives in detail. Restricting attention to local methods imposes several design constraints 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 easily be calculated 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`.

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).

## Acknowledgements

This paper 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 paper. 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 and comments from the POPL referees helped us tighten the presentation.

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

## References

[AN91] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science. Springer-Verlag, August 1991. Also available as MIT CSG Memo 329, June 1991.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press, 1993.

[Boe85] Hans-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345. IEEE, October 1985.

[Boe89] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 192–206, Portland, OR, June 1989.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer-Verlag, 1991. An earlier version appeared as DEC Systems Research Center Research Report #45, February 1989.

[Car93] Luca Cardelli. An implementation of $F_{<:}$. Research report 97, DEC Systems Research Center, February 1993.

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

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

[Com94] Adriana B. Compagnoni. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic*, September 1994. 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".

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

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, September 1996. MIT Press.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM SIGPLAN Notices*, 32(5):235–248, May 1997.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

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

[GR97] Jaques Garrigue and Didier Rémy. Extending ML with semi-explicit polymorphism. In Martín Abadi and Takayasu Ito, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS), Sendai, Japan*, pages 20–46. Springer-Verlag, September 1997.

[Hue75] Gérard Huet. A unification algorithm for typed $\lambda$-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[JG89] James W. O'Toole Jr. and David K. Gifford. Type reconstruction with first-class polymorphic values. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 207–217. ACM Press, June 1989.

[JW95] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1411–1430, September 1994. 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".

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

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Principles of Programming Languages (POPL)*, 1997. A preliminary version appeared as Technical Report, 26/96, University of Karlsruhe, July 1996.

[Per90] Nigel Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1990.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163. ACM Press, July 1988. Also available as Ergo Report 88-048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified meta-programming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Pfe93] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

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

[PL91] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic $\lambda$-calculus. *Theoretical Computer Science*, 89(1):137–159, 21 October 1991. A preliminary version appeared 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.

[Pot97] Francois Pottier. Simplifying subtyping constraints. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1997.

[PS94] Benjamin Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997.

[PT97a] Benjamin C. Pierce and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, 1997.

[PT97b] Benjamin C. Pierce and David N. Turner. Local type inference. Technical Report 493, Computer Science Department, Indiana University, 1997.

[PT97c] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University,

1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Rém94] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346, Sendai, Japan, April 1994. Springer-Verlag.

[Rey74] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[RV97] Didier Rémy and Jérôme Vouillon. 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*, pages 40–53, Paris, France, January 15–17, 1997. ACM Press. Full version to appear in *Theory and Practice of Object Systems, 1998.*

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, January 1997. Full version to appear in *Theory and Practice of Object Systems, 1998.*

[Ste97] Martin Steffen. PhD thesis, Universität Erlangen-Nürnberg, 1997. Forthcoming Ph.D. thesis.

[TS96] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer Verlag, September 1996.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

[Wan94] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 97–120. The MIT Press, 1994.

[Wel94] J. B. Wells. Typability and type checking in the second-order $\lambda$-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.

## 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 a HOT 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). 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 a HOT programming style (the 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++ or Pizza) 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.

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.

We chose the Objective Caml compiler as our experimental tool, because the front end is quite easy to understand and modify.[2] 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.[3]

We measured several publically 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 HOT 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[4] performed during typechecking: 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.

| | var. inst. | constr. inst. | comp. |
|---|---|---|---|
| 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, every 8 lines (i.e., in 13.1% of the lines). 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, 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

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

[3] The raw traces from which the tables in this section were generated are available on-line through `http://www.cs.indiana.edu/hyplan/pierce/lti-stats`.

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

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 fns | toplevel fns | other lets |
|---|---|---|---|
| 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" fn. 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 | local |
|---|---|---|
| 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 the 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.