How Good is Local Type Inference?

Haruo Hosoya
Department of CIS
University of Pennsylvania
hahosoya@linc.cis.upenn.edu

Benjamin C. Pierce
Department of CIS
University of Pennsylvania
bcpierce@cis.upenn.edu

University of Pennsylvania
Technical Report MS-CIS-99-17

June 22, 1999

Abstract

A partial type inference technique should come with a simple and precise specification, so that users predict its behavior and understand the error messages it produces. Local type inference techniques attain this simplicity by inferring missing type information only from the types of adjacent syntax nodes, without using global mechanisms such as unification variables. The paper reports on our experience with programming in a full-featured programming language including higher-order polymorphism, subtyping, parametric datatypes, and local type inference. On the positive side, our experiments on several nontrivial examples confirm previous hopes for the practicality of the type inference method. On the negative side, some proposed extensions mitigating known expressiveness problems turn out to be unsatisfactory on close examination.

1 Introduction

It is widely believed that a polymorphic programming language should provide some form of type inference, to avoid discouraging programming by forcing them to write or read too many type annotations. Unfortunately, in emerging language designs combining polymorphism with subtyping, complete type inference can be difficult [AW93, EST95, JW95, TS96, SOW97, FF97, Pot97, Pot98, etc.] or even hopeless [Wel94]. Such languages, if they are to provide any type inference at all, must content themselves with incomplete but useful partial type inference techniques [Boe85, Boe89, Pfe88, Pfe93, Car93, Nor98, Seq98, etc.].

For a partial type inference scheme to be useful, programmers must be able to understand it. The behavior of the inference algorithm should be specified in a simple and clear way, so that the users can predict its behavior and understand the causes of type errors. These considerations led Pierce and Turner to propose a class of *local type inference* schemes [PT98, PT97] that infer missing type information only from the types of adjacent syntax nodes, without resorting to global mechanisms such as unification variables.

Previous papers on local type inference predicted that it should behave fairly well in practice, based on a statistical analysis of a large body of existing polymorphic code in O'Caml and a few small examples illustrating its "feel." However, at least two questions remain: Does it really work smoothly for bigger programs? and Could we do even better? The goal of this work is to answer these questions empirically. To this end, we have designed and implemented a prototype language system with several features needed for practical programming. This paper reports the preliminary results of our experiments. We give a positive response to the first question above, but a somewhat negative response to the second.

In exploring the first question, we will be interested in two types of programs: some written in an ML-like style that makes intensive use of polymorphism and relatively little use of subtyping, and some written in an object-oriented style using both subtyping and polymorphism. In Section 3, we present extended examples in these two styles, indicating where local type inference scheme works well and where not. We find that a large proportion of the type annotations in both styles are inferred.

Turning to the second question, we focus on two (already known) kinds of situations where local type inference does *not* work so well, and consider some simple extensions that we had earlier hoped would handle some common cases better.

Hard-to-synthesize arguments The local type inference algorithm proposed by Pierce and Turner relies a combination of two techniques: (1) *local type argument synthesis* for polymorphic applications, and (2) *bidirectional propagation* of type information in which the types of some phrases are *synthesized* in the usual way, while other phrases can simply be *checked* to belong to an expected type that is determined by their context. When a polymorphic function is given an anonymous function as argument, we have a situation where either one of these techniques can be used to infer some type annotations, but not both at the same time. For example, in the term `map (fun x→x) [1,2,3]`, we cannot infer both the type arguments to `map` and the annotation on the function parameter `x` for the following reason. We take the simple approach that all the the arguments' types must be determined *before* calculation of any missing type arguments. This means that the type of the anonymous function must be synthesized with the concrete type of its parameter not available from the context. This synthesis immediately fails, since the parameter is not annotated with its type. A "bare function" like `fun x→x` is an example of a phrase for which we can *never* synthesize a type. We call such terms *hard to synthesize*.

No best type argument In local type argument synthesis, there may be more than one possible choice for a missing type argument. In such a case, we must try to find a *best* choice among the possibilities. For example, if we create an empty list by `nil unit`, there is no clue to determine the type argument to `nil`. Type argument synthesis fills the minimal type `Bot` as the type argument so the result type becomes `List(Bot)`. If the intended type is `List(Int)`, for example, `List(Bot)` can safely be promoted to the intended type. In general, we fill in missing type arguments so as to minimize the result type of the whole application. However, sometimes (in particular, when a missing type argument appears both covariantly and contravariantly in the result type) no best choice exists and type argument synthesis fails. We will see that this situation often arises in programming with parametric objects.

We had originally hoped that these problems could be solved by simple extensions of the basic techniques of local type inference: the first problem could be addressed by *avoiding hard-to-synthesize arguments* and determining type arguments from the rest of the arguments, while the second could be addressed by *taking non-best type arguments*. In Section 4, we examine these ideas more closely. Unfortunately, they turn out to be unsatisfactory. Specifically, the first substantially complicates the specification of type inference, while the second can lead to situations that we believe will be counterintuitive for the user.

Despite these limitations, our own conclusion is that the number of type annotations that can be inferred is large enough (and the ones that cannot be inferred are predictable enough) to make local type inference a useful alternative to completely explicit typing. Our main aim, though, is to give readers enough information to judge this for themselves.

2 Language Overview

To experiment with type inference for real programs, we need something close to a full-scale programming language. We use a homebrew language in the style of core ML, with some significant extensions: subtyping, impredicative polymorphism, and local type inference. Like ML, our language includes datatypes and simple pattern matching, but these mechanisms are somewhat different in detail than their analogues in ML, since here they must interact with subtyping. This section describes these extensions. All the displayed examples in this section and Section 3 have been checked mechanically by a prototype implementation.

2.1 Language Features

This subsection describes our language in the *explicitly typed* form, in which all type annotations are explicitly given.

Datatypes The syntax of datatype definitions in our language is almost the same as in ML, except that, in the case of *parametric* datatype definitions (i.e., definitions of type operators), kind annotations are required for all type parameters. As an example, the `List` datatype is defined as follows:

```
datatype List (X : *) = #nil of Unit | #cons of X * List X
```

The parameter `X` is given the kind `*`, meaning that it ranges over proper types (as opposed to type operators). (Constructors are always preceded by a hash in our concrete syntax, to simplify parsing.)

Unlike ML, our language allows the definition of datatypes with overlapping sets of constructors. For example, suppose we define the following `IList` datatype (of irregular lists, as in Scheme), sharing `#nil` and `#cons` constructors with `List` but with an extra variant `#last`:

```
datatype IList (X : *) = #nil of Unit | #cons of X * List X | #last of X
```

There is a nontrivial subtype relation between `List` and `IList`. Notice that `IList` datatype is identical to `List` datatype except that `IList` datatype has the extra variant `#last`. This means that any instance of `List` can be viewed as an instance of `IList`—the `List` instance is just restricted that it never has a `#last` cell. More precisely, `List(T)` is a subtype of `IList(T)`, for any type `T`.

In ML, the definition of `IList` shadows the constructors of `List` datatype; we cannot use the constructors of both datatypes in the same context. In our language, we may want to use both sets of constructors in the same scope. To disambiguate, each constructor is annotated with the datatype that the constructor belongs to. We call this annotation the "qualifier" of the constructor. For example, the constructor `#nil` of the `List` datatype is written `#List/cons`. In addition, because we are considering the explicitly typed form of the language, each constructor of `List` takes a type argument. For example, we can construct a `List` instance and an `IList` instance in the same scope where the definitions of both `List` and `IList` are visible, as follows.

```
let l = #List/cons [Int] (1, #List/nil [Int] unit);
let l' = #IList/cons [Int] (1, #IList/nil [Int] unit);
```

Each constructor is given either a `List` or `IList` qualifier and takes `[Int]` as a type argument. (Obviously, these annotations are somewhat awkward; type inference can often infer them, as we will see below.)

Destructors (or pattern matching), on the other hand, do not need qualifiers since the type of the term to be tested give us all the information we need.[1] For example, we can write the following pattern match.

```
let b = match l with
          #nil _ → false
        | #cons _ → true;
```

Note that the type `List(Int)` of `l` determines what variants should be listed in the pattern match.

In the explicitly typed language, the definition of a polymorphic function must explicitly declare its type parameter, and an application of the function takes a type argument. For example, we can define the following function `ilength` (meaning the "length" of a given irregular list).

```
let rec ilength [X] (l : IList(X)) : Int =
  match l with
    #nil _ → 0
  | #cons (_,l) → plus 1 (ilength [X] l)
  | #last _ → 1;
```

The function declares its type parameter as `[X]`. We can apply the function to the `IList` instance $\mathtt{l}'$ (defined above), as follows.

```
let len' = ilength [Int] l';
```

[1]This suggests that "pattern compilation" in our language will be slightly different from ML implementations. In an ML implementation, a variant appearing in a given pattern determines what datatype the pattern will match, whereas we need to know the type of the tested term.

(Again, we will see how local type inference eliminates such a type argument.)

In our language, unlike ML, a recursive call to a polymorphic function is polymorphic by default. Notice that, in the body of `ilength` function, the recursive call to `ilength` takes the type argument `[X]`. This design choice is natural here, since the definitions of polymorphic functions explicitly declare their type parameters.

Finally, the function `ilength` can also take the `List` instance `l` (defined above)

```
let len = ilength [Int] l;
```

since we have the subtyping relation `List(Int) <: IList(Int)`.

Polarities Just as the built-in type constructor $\rightarrow$ for function types is covariant in the domain and contravariant in the codomain, user-defined type constructors like `List` have a "variance" (we often use the term *polarity*) for each of their arguments. For example, `List` is covariant in its single argument.

Polarities play an important role in the subtyping relation. For example, since `List` is covariant, if we assume that `Int <: Real`, then we may expect that `List(Int) <: List(Real)`.

The covariance of `List` can be verified by looking at its definition and checking that the parameter `X` appears only in covariant positions. We require that each parameter to a datatype definition be annotated with its polarity. The default polarity is "invariant," so the definitions of `List` and `IList` as we have written them will *not* yield covariant operators. To obtain the covariant versions, we add the polarity annotation `+`:

```
datatype List (X : +*) = #nil of Unit | #cons of X * List X
```

One important reason for writing polarity annotations explicitly is that we may need to use the polarity information in a scope where the actual definition of the datatype is not available. Such a situation may arise, for example, when a functor takes as argument another module that defines a datatype. For example, we might abstract a module over a collection datatype `Col` as follows.[2]

```
module MyModule (C : COLLECTION) = struct
  ...
  let printIt (l : C.Col(Int)) (printReals : C.Col(Real)→Unit) =
    printReals l;
  ...
  end;
```

In order to apply `printReals` to `l`, we need the subtyping relation `C.Col(Int) <: C.Col(Real)`. Therefore we need to give the polarity of `C.Col` in the `COLLECTION` signature:

```
signature COLLECTION = sig
  ...
  type Col (X : +*);
  ...
  end;
```

In the case that the body of a datatype is visible, we could infer its polarity, but this is impossible when not visible. This means that the user must be aware of polarity. Furthermore, even if users do not explicitly make use of subtyping, polarity information is indispensable for successful local type inference. (More specific discussion will appear in the next section.) It may be regarded as a drawback of local type inference that it requires the extra complication of polarities.[3]

[2]This example is hypothetical since our current implementation does not support modules yet. We can obtain similar effects using higher-order polymorphism, but some features of module systems cannot be encoded in this way.

[3]A further complication in our system is that polarity information is integrated into the kind system. That is, covariant type operators have kind `+*→*`, contravariant ones have kind `-*→*`, and so on. However, if we restrict datatypes to take only proper types and quantification to be first-order, the system may be somewhat more user-friendly since kind annotation is never necessary, although polarity annotation is still necessary.

2.2 Local type inference

The local type inference scheme of Pierce and Turner [PT98] consists of two techniques: local type argument synthesis and bidirectional propagation. In this subsection, we briefly review these techniques and see how they work in the above examples.

When the type argument is missing at an application of a polymorphic function, local type argument synthesis infers it from the types of the adjacent value arguments. For example, when an identity funcion `id` is used without a type argument by writing `id 1`, the missing type argument `Int` is inferred from the type `Int` of the value argument `1`.

With bidirectional propagation, type inference operates at any given moment either in synthesis mode or in checking mode. Synthesis mode corresponds to the usual type checking, where type information is propagated in a bottom-up way—that is, the type of a term is determined from the types of its subterms. On the other hand, checking mode propagates type information top-down—that is, a term is checked to have an expected type that is determined by the context. For example, for the term `(1,2)`, synthesis mode calculates the type `Int`×`Int` from the types of the subterms `1` and `2`, while checking mode propagates the expected type `Int`×`Int` that is given by the context, verifies that the outermost constructor `(`·`,`·`)` of the term matches the outermost constructor × of the expected type, and finishes by processing the subterms `1` and `2` in checking mode, with the expected type `Int` for each.

In order to give a visual indication of how often type inference succeeds or fails, we will indicate success and failure from now on by check marks ✓ and cross marks ×, respectively, in the right-hand margin. We give either a check or a cross for each type inference "site": each datatype constructor, anonymous function, or polymorphic function application. When there are multiple type inference sites on the same line, the corresponding marks appear in the same order. The type annotations on parameters of top-level function definitions are not considered to be "type inference sites." The local type inference technique is not capable of inferring these annotations, but we believe that they constitute good program documentation and improve readability.

In the examples in the previous subsection, local type argument synthesis can infer all the type arguments in the applications of the `ilength` function. Therefore we can drop them:

```
let len = ilength l;                                        ✓
```

Here, the type argument is inferred to be `Int` because the argument `l` has type `List(Int)`. Likewise, the type arguments to the datatype constructors can be dropped:

```
let l = #List/cons (1, #List/nil unit);                     ✓✓
```

In this case, the type argument to `#List/cons` is inferred to be `Int`, but the argument to `#List/nil` is inferred to be `Bot` because the type of the argument `unit` is not relevant to the type argument. (A more detailed explanation of how type arguments are determined will be given in the next section.)

With bidirectional propagation, we can do even better. In the definition of `l`, the top-level type annotation on `l` is not available, so type inference must proceed on each subexpression expression in synthesis mode. However, if we add the top-level type annotation `List(Int)` on `l`, type inference can switch into checking mode, propagate this information to the body, and determine the qualifiers (and also the type argument) of the constructors.

```
let l : List(Int) = #cons(1, #nil unit);                    ✓✓
```

Since checking mode has more type information to work with, it has more chance of successful type inference. This makes top-level type annotations very useful for type inference, beyond their value as (mechanically checked) documentation. Therefore, local type inference encourages the use of top-level type annotations.

3 Programming Examples

We will present two programming examples in this section. Our purpose here is to illustrate how local type inference works, show how often it successfully infers type annotations, and identify typical patterns where type inference does not work well.

The first example is a set of parsing combinators [Fok95, Hut92]. This is written in "ML style," using higher-order and polymorphic functions. The purpose is to investigate whether programming with local type inference can have a comfortableness close enough to ML programming. The second example is an "object-oriented" presentation of lists. (This example uses an encoding of objects in terms of recursive types, rather than built-in object and class primitives, but we believe that the behavior of type inference here reflects the situation in a real object-oriented language reasonably accurately.)

3.1 Parsing Combinators

Parsing combinators allow users to build a parser by combining several basic library functions. Because of extensive use of streams (i.e., lazy lists), the program is slightly complicated to write in our language. (The same program in Haskell would be much more concise.) However, it suffices for the purposes of showing how local type inference works.

We assume that we have imported list and the stream libraries in which the covariant datatypes `List` and `Stream` are provided, along with familiar functions such as `length` and `map`.

We define the input stream type `IS` and the parser type `Parser` as follows:

```
type IS = Stream Char;
datatype Parser (a : +*) = #parser of IS → List (a * IS);
```

A parser is a function that takes an input stream, parses the inputs from the stream in all possible ways, and returns a list of all the parsing results. A parsing result is a pair of a value of type `a` (a parameter to the datatype `Parser`) and the remaining input stream. The functions defined below that take or return values of `Parser` datatypes will typically be polymorphic over the parameter type `a` to `Parser`. Note that the `Parser` type constructor is covariant, as indicated by the polarity annotation `+`. This fact plays a crucial role in type inference, as we shall see below.

Here are three "primitive parsers": an always-failing parser, an always-succeeding parser, and a parser that succeeds if the first input character satisfies a given predicate.

```
let pfail [a] : Parser a =
  #parser(fun is → #nil unit);                              ✓✓✓

let okay [a] (v : a) : Parser a =
  #parser(fun is → #cons((v,is), #nil unit));              ✓✓✓✓

let sat (p : Char→Bool) : Parser Char =
  #parser(fun                                                ✓✓
    #nil _ → #nil unit                                        ✓
  | #cons th → (match th unit with
      (c,is) →
        if p c then #cons((c,is), #nil unit)                 ✓✓
        else #nil unit));                                     ✓
```

Here we can see that we successfully inferred all the qualifiers of the constructors `#parser`, `#cons`, and `#nil` and all the type annotations on the anonymous function parameters.

Let us trace how type inference proceeds for the function `pfail`. Since the declaration of `pfail` specifies the result type, type inference on the body begins in checking mode, carrying along the information that the body is expected to have type `Parser a`. From this, we can infer both the qualifier `Parser` of the constructor `#parser` and its type argument `a`. This, in turn, implies that the argument to `#parser` has type `IS→List(a*IS)`. Thus, the missing type annotation on the anonymous function's parameter must be `IS`, and the body has type `List(a*IS)`. Finally, the missing type information on the constructor `#nil` is inferred similarly.

We next define another primitive parser `tok`, which succeeds if the characters at the head of the input stream match the given token (a list of characters). (The function `eqlis` has type `List Char→IS→Bool`).

```
let tok (w : List Char) : Parser (List Char) =
  #parser(fun is →                                           ✓✓
```

```
    let n = length w in                                          ✓1
    if eqlis w (stake(is, n))                                    ✓1
    then #cons((w, sdrop(is, n)), #nil unit)                 ✓ ✓1 ✓
    else #nil unit);                                             ✓
```

The three check marks highlighted with the subscript "1" correspond to the three applications of the polymorphic functions `length`, `stake`, and `sdrop` (functions whose names begin with `s` are in the stream library). The type arguments to these applications are inferred from the adjacent value arguments. Type argument synthesis for the applications of `stake` and `sdrop` are successful since they appear in checking mode. In general, type argument synthesis in checking mode succeeds whenever there is *any* type argument that makes the application well typed. On the other hand, in synthesis mode, it succeeds if there is a *best* type argument. By best, we mean the type argument that *minimizes* the result type of the application. In the above case of the application of `length` (which appears in synthesis mode), any type argument that makes the application well typed is equally the best, since the result type is `Int`, which is constant in the type argument.

The other check marks indicate places where qualifiers of constructors and annotations on anonymous function parameters are inferred.

Now we show a number of higher-order parsing combinators, which construct parsers from other parsers. (The function `dropSpaces` has type `IS→IS`.)

```
let orels [a] : Parser a → Parser a → Parser a =
  fun #parser p1 → fun                                           ✓ ✓
    #parser p2 → #parser(fun is →                                ✓ ✓
      append (p1 is) (p2 is));                                   ✓1

let sp [a] : Parser a → Parser a = fun                           ✓
  #parser p →
    #parser(fun is → p (dropSpaces is));                         ✓ ✓

let sptok (w : List Char) : Parser(List Char) =
  sp (tok w);                                                    ✓1
```

The check marks highlighted with "1" subscripts correspond to the applications of polymorphic functions, where type argument synthesis is successful since these applications appear in checking mode. The other check marks correspond to inference situations that we have already discussed.

The following combinators allow the construction of parsers by sequencing of existing parsers, by applying a given function to the results returned by an existing parser, and by restricting the results of an existing parser to those that completely exhaust the input stream.

```
let pseq [a] [b] : Parser a → Parser b → Parser (a * b) = fun    ✓
  #parser p1 → fun                                               ✓
    #parser p2 → #parser(fun is →                                ✓ ✓
      concat                                                     ✓1
        (map                                                     ✓1
          (fun ((v1:a),(is1:IS)) →                               ×2
            map                                                  ✓1
              (fun ((v2:b),(is2:IS))→((v1,v2),is2))              ×2
              (p2 is1))
          (p1 is)));

let pam [a] [b] : Parser a → (a → b) → Parser b = fun            ✓
  #parser p → fun f → #parser(fun is→                          ✓ ✓ ✓
    map                                                          ✓1
      (fun ((v:a), (is1:IS))→                                    ×2
        (f v, is1))
      (p is));

let just [a] : Parser a → Parser a = fun                         ✓
  #parser p → #parser(fun is→                                    ✓ ✓
```

```
  filter                                                          ✓1
    (fun ((v:a),(is:IS)) →                                        ×2
      (match dropSpaces is with
        #nil _ → true
      | #cons _ → false))
    (p is));
```

The uses of the polymorphic functions `concat`, `map`, and `filter` in these definitions give rise to both ✓ and × marks—i.e., some type annotations can be inferred, but others can't.

On the positive side ($\checkmark_1$), we succeed in inferring the type arguments to every application of a polymorphic function. In the above example, synthesizing the type arguments of `concat` in `pseq`, `map` in `pam`, and `filter` in `just` succeeds because they are in checking mode. The two applications of `map` in `pseq` appear in synthesis contexts, but the inference succeeds for the following reason. The function `map` has type `All[X]All[Y](X→Y)→List X→List Y`, in which the result type `List Y` is constant in `X` and covariant in `Y`. Therefore the best type argument for `X` is any one (that makes the application well typed), and the best type argument for `Y` is the minimal one (that makes the application well typed). (E.g., in the first application of `map`, we take `(a×b)×IS` for `Y` because `Y` can range between `(a×b)×IS` and `Top`.) Here, note that the polarity information `+` on `List` is important because this makes the result type covariant. (In general, if the result type covariant, we can always find a best type argument. It is also the case when the result type is contravariant. However, if the result type is *invariant*, synthesis fails unless there is only one possible type argument. We will see in Section 3.2 an example where synthesis fails for this reason.)

On the negative side ($\times_2$), when an anonymous function appears as the argument to a polymorphic function, we fail to infer its parameters' types. This is because we take the simple approach of synthesizing the types of all term arguments before calculating the type arguments, but synthesis of bare functions (i.e., anonymous functions without type annotations on their parameters) always fails. In such a case, we must supply either the type argument or the type annotation on the anonymous function's parameter. We call terms for which synthesis always fails (such as a bare function) *hard-to-synthesize*. Unqualified constructors are also hard to synthesize and we will see below a similar problem. Note also that this problem happens whether the application occurs in checking mode or in synthesis mode.

An obvious question is whether we can make the type inference cleverer so as to somehow avoid synthesis of hard arguments and calculate the type arguments only from the types of the remaining arguments. For example, consider `filter (fun x→x) [1,2,3]`. Even if we avoid synthesis of the anonymous function, the second argument gives us enough information to determine the type argument as `Int`. We will discuss this idea in Section 4.2.

The last three combinators make use of many parsing combinators that are already defined: `many` parses sequences using an existing parser, `many1` parses nonempty sequences, and `listOf` parses sequences with separators. The function `makelist` is a helper.

```
let makeList [a] ((x : a), (xs : List a)) : List a =
  #cons(x,xs);                                                  ✓

let many [a] (p : Parser a) : Parser(List a) =
  let rec q _ : Parser(List a) =
    (orels                                                      ✓
       (pam                                                     ✓
          (pseq p                                               ✓
                (#Parser/parser                                 ×1
                  (fun (is:IS) →                                ×2
                    match q unit with #parser q → q is)))
          (makeList [a]))                                       ×3
       (okay                                                    ✓
         (#List/nil unit)))                                     ×1
  in q unit;

let many1 [a] (p : Parser a) : Parser(List a) =
  pam                                                           ✓
```

```
     (pseq p                                                  ✓
           (many p))                                          ✓
     (makeList [a]);                                          ×3

let listOf [a] [b] (p : Parser a) (s : Parser b) : Parser(List a) =
  let nonempty ((x : a),  (xs : List(b * a))) : List a =
    #cons(x, map                                              ✓✓
               (snd [b] [a])                                  ×3
               xs)
  in
  (orels                                                      ✓
     (pam                                                     ✓
        (pseq p                                               ✓
              (many                                           ✓
                (pseq s p)))                                  ✓
        nonempty)
     (okay                                                    ✓
       (#List/nil unit)));                                    ×1
```

(The local function `q` defined in `many` takes the "wild-card" parameter `_`, which is implicitly given type annotation `Top`.) It is pleasant to see that most of type arguments to the polymorphic functions are inferred. Again, the covariance of the type `Parser` is the key to success.

However, there are some noticeable failures. At $\times_1$, type inference fails to infer the constructor qualifiers. This is the same problem as before except that the hard-to-synthesize arguments are unqualified constructors here. At $\times_2$, we further need to add the type annotation on the function parameter because the function is the argument of a constructor and the type checker reaches it in synthesis mode. This reveals an unfortunate characteristic of bidirectional propagation: once we get into synthesis mode, it is not easy to get out.

At $\times_3$, we need to supply the type arguments to the polymorphic functions `makeList` and `snd`. This is because these, in turn, are given to functions that expect monomorphic types. The scheme cannot synthesize type arguments to polymorphic functions with no value arguments. (Even if we had some mechanism to "trigger" type argument synthesis here, we would still fail, because there is no way to determine the type argument locally in synthesis mode. However, if we had a similar situation in checking mode, type argument synthesis would succeed.)

3.2 List Objects

As an example of a program using *both* polymorphism and subtyping, we now present an object-oriented list data structure. This example heavily uses advanced typing features not found in ML (whereas the previous example fits within ML). Our presentation uses an "object encoding" in terms of recursive types, and so is somewhat heavier than it would be in a language with primitive objects and classes. Nevertheless, it allows us to "preview" what might happen when local type inference is used in a high-level object-oriented language. In particular, we are mainly concerned with typing issues, and in this respect we are not so far from reality.

In the "recursive encoding," an object is a record of methods. The type of an object is a record type of types for methods that may recursively refer to the object's type. In the following, we define an object type `OList` for lists.[4]

```
datatype OList (X : *) = #object of {
  null : Unit → Bool,
  length : Unit → Int,
  append : OList X → OList X,
  hd : Unit → X,
  tl : Unit → OList X,
  last : Unit → X,
```

[4] We need to define this as a datatype because this is the only way to define a recursive type. This leads to a spurious `#object` constructor appearing in many places.

```
add : X → OList X,
nth : Int → X,
take : Int → OList X,
drop : Int → OList X,
app : (X → Unit) → Unit,
map : All[Y](X → Y) → OList Y,
mapPartial : All[Y](X → Option Y) → OList Y,
find : (X → Bool) → Option X,
filter : (X → Bool) → OList X,
exists : (X → Bool) → Bool,
all : (X → Bool) → Bool };
```

The `OList` type is parametric in the type of the elements of the list. Since the parameter has no polarity annotation, the `OList` type is invariant. This is inevitable (no other annotation would be correct) because list objects have an `add` method that takes an element `X` as its argument. This fact has a negative impact on type inference, as we will see below.

Let us define a creation function for list objects. First, we need to define the data structure for internal states (intuitively, the "instance variables") of list objects:

```
datatype State (X : *) = #nil of Unit | #cons of X * OList X;
```

That is, the internal state of a list is either `nil` or a pair of an element and the next list cell. Note that this datatype is also invariant.

We define the creator function `mkolist` as follows. (For ease of explanation, we split the big function into several pieces.) The function `mkolist` takes an element type `X` and an internal state `st` and returns a list object of type `OList X`:

```
let rec mkolist [X] (st : State X) : OList X =
```

In the body, we first define local functions `getmethods` and `self`, for convenience. The function `getmethods` extracts the record of the methods from an object, while `self` constructs a list object whose internal state is `st`:

```
let getmethods (x : OList X) = match x with
  #object o → o in
let self _ : OList X = mkolist st in                         ✓
```

We then return a list object, which is a big record tagged with the `#object` constructor. The first three methods are simple. (We elide the bodies of some similar methods.)

```
#object{
  null = fun _ → match st with
    #nil _ → true
  | #cons _ → false,

  hd = fun _ → match st with
    #nil _ → error unit
  | #cons (hd,_) → hd,

  tl = ...,
```

The next several methods are recursively defined (using the `getmethods` function to invoke a method).

```
  length = fun _ → match st with
    #nil _ → 0
  | #cons (_,tl) → plus 1 (((getmethods tl).length) unit),

  nth = fun n → match st with                                ✓
    #nil _ → error unit
  | #cons(hd,tl) → if eq n 0 then hd
```

```
    else (getmethods tl).nth (minus n 1),

find = fun p → match st with                                     ✓
  #nil _ → (#none unit)                                          ✓
| #cons(hd,tl) →
    if p hd then #some hd else (getmethods tl).find p,           ✓

last = ...,
drop = ...,
exists = ...,
all = ...,
app = ...,
```

It should be noted that these methods are monomorphic. That is, the element type of a list object is instantiated at *creation* time, and we do not need to supply a type at each method invocation. This is contrast to programming with the list datatype where we need a type argument at each application of an operation to a list. This suggests that polymorphism in programs with parametric objects is coarser than in ML programming, and therefore that the impact of local type inference may be somewhat smaller.

Notice also that enclosing the record of methods with the constructor `#object` (and the fact that we know the qualifier of this constructor) allows us to proceed with the record body in checking mode. This is the key to successful inference here. If the record were in a synthesis context (e.g., if the record were first bound to a variable by `let` and then tagged with `#object` to create an object), then a lot of type annotations in the big record would remain. In particular, we would have to give explicitly all of the type annotations on the function parameters, all of the constructor qualifiers, and so forth. This suggests that a higher level object-oriented language should be designed in such a way that the method bodies can be dealt with in checking mode. (This highlights the observation that the effectiveness of local type inference depends on the surface language: the richer the surface language, the more effective type inference will tend to be.)

In the next two methods, some constructor qualifiers are not inferred because of the problem of hard-to-synthesize arguments:

```
add = fun x → mkolist (#State/cons(x, self unit)),               ✓✓×

append = fun lis → match st with                                 ✓
  #nil _ → lis
| #cons(hd,tl) →
    mkolist (#State/cons(hd, (getmethods tl).append lis)),        ✓×
```

In the next two, we encounter a new reason for failure of type argument synthesis:

```
take = fun n →                                                    ✓
  if eq n 0 then mkolist (#State/nil[X] unit)                     ✓×1
  else (match st with
    #nil _ → error unit
  | #cons(hd,tl) →
      mkolist (#State/cons(hd, (getmethods tl).take (minus n 1)))),  ✓×

filter = fun p → match st with                                    ✓
  #nil _ → mkolist (#State/nil[X] unit)                           ✓×1
| #cons(hd,tl) → if p hd
    then mkolist (#State/cons(hd, (getmethods tl).filter p))      ✓×
    else (getmethods tl).filter p,
```

Here ($\times_1$), we fail to infer the type arguments to the constructor `#State/nil`, for the following reason. Since these applications of the constructor are in synthesis contexts, we try to find the best type argument, which is impossible because the result type of the constructor is invariant in its parameter and the argument `unit` does not give any useful type information for the type argument. (In the case of the `#State/cons` constructor, even though its result is invariant, the fact that the expected type of the second argument is invariant makes the possible type argument unique.)

The next two methods are interesting because they are polymorphic:

```
    map = fun[Y] f → match st with                                          ✓
      #nil _ → mkolist (#State/nil[Y] unit)                                ✓×
    | #cons(hd,tl) →
        mkolist (#State/cons[Y] (f hd, (getmethods tl).map f)),         ✓ ×1 ✓

    mapPartial = fun[Y] f → match st with                                   ✓
      #nil _ → mkolist (#State/nil[Y] unit)                                ✓×
    | #cons(hd,tl) → (match f hd with
        #none _ → (getmethods tl).mapPartial f                              ✓
      | #some r → mkolist                                                   ✓
            (#State/cons[Y](r, (getmethods tl).mapPartial f))),            ×1✓
  };
```

Here, we need the type argument `Y` to `#State/cons` ($\times_1$). The reason is slightly different from the case of `#State/nil` above. If we drop the type argument to `#State/cons`, then for type argument synthesis, we try to synthesize the types of the arguments, one of which is an invocation of the polymorphic method `map` (or `mapPartial`). However, note that its type (i.e., the type of `(getmethods tl).map`) is `All[Y′](X→Y′)→OList Y′` (alpha-converted in order to avoid confusion), and the result type `OList Y′` is invariant in `Y′`. Moreover, the type `X→Y` of the argument `f` does not constrain the possible type arguments to be unique. Consequently, type argument synthesis fails here. (Alternatively, we can drop the type argument to the constructor `#State/cons` by supplying the type argument to the polymorphic method.)

The central reason for the failures here (and for the `#State/nil` cases above) is that there is no best type argument. This might lead us to wonder whether we could get away with taking some other (suboptimal) type argument instead of failing. We discuss this in Section 4.1.

In addition to the above creator function, we may also want a function to create an empty list.

```
let mkolist_empty [X] _ : OList X = mkolist (#State/nil [X] unit);          ✓×
```

Having defined the creator functions, let us see how they are used. Because of the invariance of `OList`, we need some type annotations at creation time—we can choose between writing

```
let l1 = mkolist_empty [Int] unit;                                          ×
```

or:

```
let l1 : OList Int = mkolist_empty unit;
```

We then use the created list object for adding, extracting, and mapping into another list object:

```
let l2 = match l1 with
  #object o → o.add 1;
let l3 = match l2 with
  #object o → plus (o.hd unit) 2;
let l4 = match l2 with
  #object o → o.map [Int] (fun x → plus x 1);                               ×✓
```

On the last line, we need to supply the type argument `Int` to `o.map` because of the invariance of the result type of the method. Alternatively, we can add type annotation at the top level and the function parameter, and remove the type argument:

```
let l4 : OList Int = match l2 with
  #object o → o.map (fun (x : Int) → plus x 1);                             ✓×
```

3.3 A Final Observation

As we mentioned in Section 2, recursion is polymorphic in our language. One disadvantage of this is that this sometimes causes a failure of inferring some type annotations even though the recursion is essentially monomorphic. Consider the following example (extracted from the list library):

```
let rec revAcc [X] (xs : List(X)) (ys : List(X)) : List(X) =
  (match xs with
    #nil _ → ys
  | #cons (x,xs) → revAcc xs (#List/cons (x,ys)));                    ✓×
```

In the second argument to the polymorphic function `revAcc`, the qualifier of `#List/cons` is not inferred, because of the problem of hard-to-synthesize arguments. We could deal with this by avoiding such arguments, as discussed in Section 4.2. However, we would not have this problem at all if the recursion were monomorphic. Therefore we might imagine circumventing such situations by some syntactic trick that allows a monomorphic recursion in the body of the polymorphic function. (We have not invented such a trick yet, however.)

4 Could We Do Better?

In the previous section, we identified several patterns of suboptimal behavior by the local type inference algorithm. In this section, we focus on two of them, the problem of "no-best-type-argument" and the problem of "hard-to-synthesize arguments," and attempt to improve the algorithm's behavior in some obvious ways. Unfortunately, the improvements do not turn out to be very satisfactory.

4.1 Taking Non-best Type Argument

We have seen that type argument synthesis fails when the result type is invariant in the missing type parameter and the value arguments do not determine the possible type argument uniquely. For example, suppose using the reference creation function `ref` of type `All[X]X→Ref(X)`, and we create a reference cell by writing `ref 1`, without a type argument. Since the `Ref` constructor is invariant and the missing type argument here might be any type greater than `Int`, there is no best type argument (e.g., `Ref(Int)` and `Ref(Real)` are both possible, but incomparable). Thus type argument synthesis fails.

One obvious attempt at a solution is to allow the type checker to choose some non-best type argument. It is clear that adopting such a solution is a bit dangerous (since it means that the type checker will sometimes infer a result type for an application that cannot be promoted to the type the user had in mind), but we might expect that it would work in practice since in some clearly identifiable cases, there is a reasonable (though non-best) guess. For example, if we create a reference cell by writing `ref 1`, since the value argument is an integer, one reasonable guess is to choose the minimal type argument `Int`.

Unfortunately, there appear to be many cases—even in fairly similar expressions—where the value arguments do not provide enough information to make a good guess at missing type parameters. For example, suppose we create an empty list object of the invariant `OList` datatype, using the `mkolist_empty` creator function from Section 3.2.

```
let l1 = mkolist_empty unit;
```

Since there is no clue what the rest of computation expects, the best we could do is to guess `Bot` or `Top`. However, both guesses are bad. If we choose `Bot`, we immediately run into a problem when adding an element:

```
let l2 = match l1 with
  #object o → o.add 1;
```

(We fail at `o.add 1` because `o.add` expects `Bot` type.) On the other hand, if we choose `Top` as the type argument, since `o.add` expects `Top` type, it will go through. But the result type is now `OList(Top)`, and we will have a problem when extracting an element:

```
let l3 = match l2 with
  #object o → plus (o.hd unit) 2;
```

(The result type of `o.hd unit` is `Top` and therefore cannot be applied to `plus`.) Because taking non-best type argument does not work even in such a simple and typical situation, we do not think that this is a good way to solve the problem considered here.

The designers of GJ have proposed a different solution to the problem of local type argument synthesis in the presence of invariant result types [BOSW98]. Roughly, it involves extending the type system with an "indeterminate type," written `*`, which can be promoted to any type, and choosing `*` for the missing type parameters in cases like `mkolist_empty unit`. (In effect, `*` can be thought of as simple form of unification variable.) To maintain soundness, some rather delicate side conditions are needed.

4.2 Avoiding hard-to-synthesize arguments

Type inference fails when a polymorphic function is applied to a hard-to-synthesize argument, such as a bare function or an unqualified constructor, without any type arguments. For example, consider

```
let l0 : List Int = #cons(1, #nil unit);
let l : List Int = map (fun x → plus x 1) l0;
```

(where `map` has type `All[X]All[Y](X→Y)→List(X)→List(Y)`). When type inference encounters the application of `map` (in checking mode, with type `List(Int)` expected), it notices that a type argument is omitted and begins synthesizing the types of the value arguments, as preparation for calculating the omitted type argument. Unfortunately, the first thing it encounters is the anonymous function, where it fails.

We might wonder whether we could simply avoid synthesizing the type of the anonymous function, calculate the missing type argument from the type of the other argument `l0`, and finish by processing the anonymous function in checking mode. For this example, this works well, since the type `List Int` of `l0` tells what we expect as the type argument for `X`.

However, this scheme turns out to be unsatisfactory, firstly because we need to substantially complicate type inference specification, and secondly because this scheme still tends to fail.

We have two sources of complication. The first is that we need some mechanism to decide which arguments to avoid. The most obvious approach is a simple syntactic analysis to detect hard-to-synthesize terms:

$$\mathtt{h} ::= \mathtt{fun(x)e} \mid \mathtt{fun(x{:}T)h} \mid \mathtt{\#l\ e} \mid \ldots$$

That is, hard-to-synthesize terms are bare functions, non-bare functions whose bodies are hard to synthesize, unqualified constructors, and so on. This approach is slightly ad-hoc, but seems reasonable in practice and easy to understand for users.

The second source of complication is that we need a more complicated scheme for choosing a best type argument. Let us show how a naive use of the previous scheme of local type argument synthesis does not work. In the above example, when type inference in checking mode processes the application, it is given the expected type `List(Int)` (which is propagated from the top-level annotation), and also obtains the type `List(Int)` of the second actual argument `l0` (which was not avoided). From these, we calculate the constraints on the first type argument `X` and the second type argument `Y` as follows.

$$\mathtt{Int} <: \mathtt{X} <: \mathtt{Top} \quad \text{and} \quad \mathtt{Bot} <: \mathtt{Y} <: \mathtt{Int}$$

In the previous scheme, checking mode chooses arbitrary types for the type arguments. But it does not work. If we choose `Int` for `X` and `Bot` for `Y`, for example, then we would propagate the type `Int→Bot` for the anonymous function, which will fail at the body `plus x 1`.

It turns out that a best type argument is one that *maximizes* the expected type corresponding to the avoided argument, because the maximal type can always be *demoted* to the intended type. (In synthesis mode, we need to *minimize* the result type at the same time.) In the above example, we determine both type arguments `X` and `Y` to be `Int` since these maximize the first expected type `X→Y`.

The second reason why avoiding hard-to-synthesize arguments is unsatisfactory is that type inference tends to fail anyway, because it does not receive enough type information from the remaining arguments. For example, consider the following piece of code (identical to the above except for absence of top-level annotation):

```
let l = map (fun x → plus x 1) l0;
```

Again, we need to maximize the first parameter type `X→Y`, but this time we need to minimize the result type `List(Y)`, too. Therefore we essentially need to minimize `(X→Y)→List(Y)`. However, since this type is invariant in `Y` and `Y` is not constrained at all, we cannot find a best type for `Y`, and we fail.[5]

One can argue that type annotations on *all* function parameters are useful documentation. It may be better to stick with a language design that assumes type annotations on function parameters, rather than trying to come up with complicated schemes for inferring some of them.

5 Conclusion

We have examined the usefulness of local type inference scheme by attempting to address two questions: (1) Does the scheme work well in practice? And (2) could we do better? Our response to the first question is positive. We experimented on several examples in both ML style and object-oriented style, confirming previous expectations by observing that a fairly large part of type annotations were inferred.

We are negative, though, on the second question. We have focused on two known problems—the problem of hard-to-synthesize arguments and the problem of no best type argument—tried to solve these problems by simple extensions. However, we have seen that these extensions (the best ones we could think of) do not work very well. From these lessons, we think that rather than making type inference cleverer, we should try to design high-level syntactic constructs that are natural for users and also aid type inference in an effective way. A key property of such constructs is to keep the bidirectional propagater in checking mode as much as possible.

A final idea might be that restricting the type system may make type inference easier. Restricting ourselves to the F_3 kind system does not seem to help much (since the type inference scheme for $F^{\omega}_{\leq}$ used here is not much more complex than the previously proposed scheme for $F_{\leq}$), but restriction to predicative polymorphism might, in view of recent work on type inference for predicative calculi without subtyping. However, from the expressiveness point of view, we believe that impredicativity is important for programming with polymorphism and objects. (Recall that our list objects had polymorphic methods. Since type variables can range over types like `OList(T)`, it appears that impredicativity is inescapable in this setting.)

Acknowledgments

Haruo Hosoya was supported by University of Pennsylvania and by the Japan Society for the Promotion of Science. Haruo Hosoya thanks Professor Yonezawa for his encouragement. Benjamin Pierce was supported by the University of Pennsylvania and by NSF grant CCR-9701826, *Principled Foundations for Programming with Objects*. Conversations with Robert Harper and Karl Crary were helpful in polishing this paper.

References

[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.

[5] One might wonder what happens if we take a non-best type argument here. For the same reason described in Section 4.1, taking non-best type argument may conflict with the user's intention. In the above example, since `Y` is not constrained, we could choose either `Bot` or `Top` for `Y`. However, either choice is bad. If we choose `Bot`, then this causes checking of the body `plus x 1` of the anonymous function with `Bot`, which fails. If we choose `Top`, then since the result type will become `List(Top)`, we will fail at some point in the rest of the program.

We feel that this situation is even worse than in Section 4.1, because the cause of the problem is much more complicated—involving avoidance of a hard argument, maximization of the expected type, and taking non-best type argument. Even if we precisely specify the decision of type argument synthesis, it is far from easy for the user to figure out the cause. This will clearly break our slogan "simple and clear method".

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Proceedings of OOPSLA'98*, 1998.

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

[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.

[Fok95] Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced functional programming, Baastad Summerschool Tutorial Text*, volume 925 of *LNCS*, pages 1–23. 1995.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of functional programming*, 2:323–343, 1992.

[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.

[Nor98] Johan Nordlander. Pragmatic subtyping in polymorphic languages. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

[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.

[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.

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

[Pot98] François Pottier. A framework for type inference with subtyping. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

[PT97] Benjamin C. Pierce and David N. Turner. Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University, 1997.

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

[Seq98] Dilip Sequeira. *Type Inference with Bounded Quantification*. PhD thesis, University of Edinburgh, 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.*

[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.

[Wel94] J. B. Wells. 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)*, pages 176–185, 1994.