CS 510: Programming Languages
Assignment #6

Due: Monday, 8 December, 2003

1 Introduction

In this assignment, we will study *bi-directional type inference*, which is a partial type inference technique that allows many type annotations to be inferred automatically from the context of a term. Unlike ML-style type inference, which only works for languages with predicative prenex-polymorphism, bi-directional type inference scales to much richer languages, including languages with general predicative polymorphism and subtyping.

2 Type inference for PolyMinML

Consider the following example:

```
fun f(x : int) : unit -> unit is
   fun g(y : unit) : unit is y end
end
```

In this example, the type annotations for the argument and return type of `g` are redundant; because f specifies its return type, we know what the type of g must be. This frequent repetition of type information makes using plain PolyMinML tedious. Take a look at how we might write and use the polymorphic identity function.

```
if ((typefun t. fun f(x : t) : t is x end) [ bool ] true) then 1 else 0
```

Here, the instantiation of the polymorphic function at type bool is redundant; we know that the result must be of type bool, since it is the conditional in an `if` expression. Thus, some expressions need type annotations, and some do not. The essence of our type-checking algorithm will be to make a syntactic distinction between expressions whose type can be inferred, and those whose types can be checked (given some context about what their type *should* be).

The intuition is that *checkable* expressions appear only where their type can be determined by context, and so they don't need type annotations. *Inferable* expressions are what we've primarily used in previous type checkers. These have a unique type which can be determined in a particular variable context Γ.

With two different kinds of expressions, we have two forms of typing judgments:

$$\overline{\Delta, \Gamma \vdash c \downarrow \tau}$$

$$\overline{\Delta, \Gamma \vdash i \uparrow \tau}$$

The first says that given a type variable context Δ and an expression variable context Γ, *as well as the target type* τ, we can verify that the checkable expression c has the given type τ.

The second says that given similar contexts Δ and Γ, we can deduce the unique type τ for the inferable expression i.

To recap: We separate expressions into two categories, checkable and inferable. Then we have two separate judgments: one which says that given a type, we can verify a checkable expression has that type, and one which allows us to deduce the type of an inferable expression. give a precise description of the abstract syntax and the algorithm we use to check checkable expressions and infer the types of inferable expressions.

2.1 PolyMinML Abstract Syntax

Fill in the definition of the inferable terms and the checkable terms. Remember, these definitions can be mutually recursive. In other words, checkable terms can have inferable terms inside them and vice versa. The "goal" is to minimize annotations and the need for explicit typing information. Therefore, if you have a choice between making the term checkable and inferable, you should always make it inferable because less typing information is needed. For example, the integer n should be classified as inferable because we know that n's type is always `int`. It is never to our advantage to classify it as checkable. On the other hand, if, during the course of processing an expression, one has in hand enough information to be able to check the subexpressions, it is better to make them checkable than inferable. For example, we will classify `fun` f `(`x `:` τ_d`) :` τ_r `=` c as inferable, since τ_d and τ_r give us enough information to uniquely determine the type of the function. However, the body of the function should be a checkable term c since we know its type must be τ_r and we can pass τ_r to the inference algorithm when processing c. Hence it is not generally the case that the subterms of an inferable term need to be inferable themselves (make them checkable whenever possible). Nor is it the case that the subterms of a checkable term are always checkable (make them checkable whenever possible, but sometimes you do not have sufficient information).

Here is a partial definition of the inferable and checkable terms. You need to fill in further details. In particular, you need to fill in the definitions of if statements (two of them: `if` for inferable and `cif` for a corresponding checkable if statement), function application (two possibilities, one checkable and one inferable) and type application (two possibilities; you do not want the programmer to have to write down the type to be instantiated if possible).

```
inferable i ::= ()
          | x
          | n
          | true
          | false
          | c1 + c2
          | fun f (x : τd) : τr = c
          | typefun t. i
          | ...

checkable c ::= I (i)
          | cfun f(x) = c
          | ctypefun t. c
          | ...
```

All of the inferable expressions should be familiar. For each one, we can determine its unique type using the standard typing rules. Note the places where checkable expressions c appear. Since we know the types of the arguments to primops (i.e., +), those expressions are also cs.

Any inferable expression is checkable, since we can just ignore the context and get the type using the inference algorithm (however, it is still necessary to distinguish an inferable expression from an inferable expression *as a checkable expression*). The form `I ( i )` means an i converted into a c. The `cfun` expression is checkable: given the type it is supposed to have (which must be a function type), we can type-check the function as we would for a standard annotated `fun`. `ctypefun` is checkable if its body is checkable; we simply wrap the type of the body in a $\forall$ type. All of these strategies are made precise in the next section.

2.2 PolyMinML Typing Rules

Inferable Expressions

$$\frac{}{\Delta, \Gamma \vdash \texttt{()} \uparrow \texttt{UNIT}} \quad \textit{(unit)}$$

$$\frac{}{\Delta, \Gamma \vdash n \uparrow \texttt{INT}} \quad \textit{(int)}$$

$$\frac{}{\Delta, \Gamma \vdash \texttt{true} \uparrow \texttt{BOOL}} \quad \textit{(true)}$$

$$\frac{}{\Delta, \Gamma \vdash \texttt{false} \uparrow \texttt{BOOL}} \quad \textit{(false)}$$

$$\frac{\Gamma(x) = \tau}{\Delta, \Gamma \vdash x \uparrow \tau} \quad \textit{(var)}$$

$$\frac{\Delta, \Gamma \vdash c_1 \downarrow \texttt{INT} \quad \Delta, \Gamma \vdash c_2 \downarrow \texttt{INT}}{\Delta, \Gamma \vdash \texttt{+(}c_1, c_2\texttt{)} \uparrow \texttt{INT}} \quad \textit{(plus)} \text{ etc.}$$

$$\frac{\Delta \vdash \tau_1 \ \mathsf{ok} \quad \Delta \vdash \tau_2 \ \mathsf{ok} \quad \Delta, \Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash c \downarrow \tau_2}{\Delta, \Gamma \vdash \texttt{fun} \ f\texttt{(}x : \tau_1\texttt{)} : \tau_2 = c} \quad \textit{(fun)}$$

$$\frac{\Delta[t], \Gamma \vdash i \uparrow \tau}{\Delta, \Gamma \vdash \texttt{typefun} \ t.i \uparrow \forall t.\tau} \quad \textit{(forall)}$$

Checkable Expressions

$$\frac{\Delta, \Gamma \vdash i \uparrow \tau' \quad \tau = \tau'}{\Delta, \Gamma \vdash \texttt{I(}i\texttt{)} \downarrow \tau} \quad \textit{(inferable)}$$

$$\frac{\Delta, \Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash c \downarrow \tau_2}{\Delta, \Gamma \vdash \texttt{cfun} \ f(x) = c \downarrow \tau_1 \to \tau_2} \quad \textit{(cfun)}$$

$$\frac{\Delta[t], \Gamma \vdash c \downarrow \tau}{\Delta, \Gamma \vdash \texttt{ctypefun} \ t.c \downarrow \forall t.\tau} \quad (forall_c)$$

These rules reflect the observations we made in the last section.
You need to fill in the missing rules.

2.3 Implement Bi-directional Type Checking

The rules given in the previous section are suggestive of an algorithm, consisting of two mutually recursive functions, `check` and `infer`. Both take a set of type variables (Δ), and a variable context (Γ). `infer` then takes an i and returns its unique type τ in that context (if it exists). `check` takes a c and a type τ and verifies that it is compatible with that type.

Complete the PolyMinML type-checker by implementing the functions `check` and `infer` in the file `typing.sml` and adding to the abstract syntax definitions and editing the substitution and evaluation functions in the `polyminml.sml` file. These functions should meet the following specification:

```
check : (var → bool) → (var → typ) →
          cexp → typ → unit
```

Such that `check` Δ Γ c τ returns unit if the type is consistent, or raises the exception `TypeError` with an informative message otherwise.

```
infer : (var → bool) → (var → typ) →
          iexp → unit
```

Such that `infer` Δ Γ i returns the unique type τ of i in that context, or raises the exception `TypeError` with an informative message if no such τ exists.

Hints

- Avoid unnecessarily inefficient or convoluted code, and document non-trivial invariants/preconditions.
- Make sure to be careful about type variables, especially concerning capture and alpha-equivalence ($\forall t.t \rightarrow t$ and $\forall u.u \rightarrow u$ are the *same type*).
- You will probably want to make use of the functions defined in `types.sml` and documented in `types.sig`. However, you only need to modify `typing.sml` and `polyminml.sml` for your solution to this part of the assignment.
- We haven't given you a nice parser and pretty printer for this assignment. In order to test your type checker, you need to construct test expressions directly in ML using the datatype declared in the `polyminml.sml` file. Make up some test cases and put them in the file `testcases.sml`. Make up tests that both pass and fail. Test one of these expressions with, for example:

  ```
  Typing.typecheck Test.t1;
  ```