CS 510: Programming Languages
Assignment #5

Out: Tuesday, 19 November, 2002
Due: Wednesday, 27 November, 2002

1 Introduction

In this assignment, you will write a type-checker for a version of the PolyMinML language. This type checker will perform *bi-directional type inference*, allowing 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:

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

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

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. The next two sections 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

$$
\begin{array}{lll}
\texttt{inferable}\ i & ::= & \texttt{()} \\
& \mid & x \\
& \mid & n \\
& \mid & \texttt{true} \\
& \mid & \texttt{false} \\
& \mid & \texttt{primop}\ (c_1,\ c_2,\ \ldots) \\
& \mid & i\ c \\
& \mid & \texttt{fun}\ f\ (x\ :\ \tau_d)\ :\ \tau_r\ \texttt{is}\ c\ \texttt{end} \\
& \mid & \texttt{if}\ c_b\ \texttt{then}\ i_t\ \texttt{else}\ i_f\ \texttt{fi} \\
& \mid & i\ [\ \tau\ ] \\
& \mid & \texttt{typefun}\ t.\ i \\
\\
\texttt{checkable}\ c & ::= & \texttt{I}\ (i) \\
& \mid & \texttt{cfun}\ f(x)\ \texttt{is}\ c\ \texttt{end} \\
& \mid & \texttt{cif}\ c_b\ \texttt{then}\ c_t\ \texttt{else}\ c_f\ \texttt{fi} \\
& \mid & i\ [] \\
& \mid & c\ i \\
& \mid & \texttt{ctypefun}\ t.\ c
\end{array}
$$

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. If we know the type of a function (because it is inferable), we know what the type of the argument must be, so the argument need only be a *checkable* expression. The body of a function is also checkable, since its type is given in the function expression. Since we know the types of the arguments to primops, those expressions are also cs. Finally, we know that the conditional expression in an `if` must be a boolean, so that too is checkable.

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`. The `cif` expression can be checked as well, since we know the condition must be of type `BOOL`, and we know that the types of the branches must match the type of the whole expression. A checkable expression applied to an inferable expression is checkable: we know the return type of the function (it is the type we are checking against), and we know the domain type (it is the type of the argument). `ctypefun` is checkable if its body is checkable; we simply wrap the type of the body in a $\forall$ type. Finally, we are able to instantiate a polymorphic inferable expression at an unspecified type, provided we know what the type of the instantiation is supposed to be. All of these strategies are made precise in the next section.

2.2 PolyMinML Typing Rules

Inferable Expressions

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

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

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

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

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

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

$$\frac{\Delta, \Gamma \vdash i \uparrow \tau' \to \tau \quad \Delta, \Gamma \vdash c \downarrow \tau'}{\Delta, \Gamma \vdash i\ c \uparrow \tau} \quad \textit{(app)}$$

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

$$\frac{\Delta, \Gamma \vdash c \downarrow \texttt{BOOL} \quad \Delta, \Gamma \vdash i_t \uparrow \tau \quad \Delta, \Gamma \vdash i_f \uparrow \tau}{\Delta, \Gamma \vdash \texttt{if}\ c\ \texttt{then}\ i_t\ \texttt{else}\ i_f\ \texttt{fi} \uparrow \tau} \quad \textit{(if)}$$

$$\frac{\Delta \vdash \tau\ \mathsf{ok} \quad \Delta, \Gamma \vdash i \uparrow \forall t.\tau'}{\Delta, \Gamma \vdash i[\tau] \uparrow [\tau/t]\tau'} \quad \textit{(instantiate)}$$

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

Checkable Expressions

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

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

$$\frac{\Delta, \Gamma \vdash c \downarrow \tau' \to \tau \quad \Delta, \Gamma \vdash i \uparrow \tau'}{\Delta, \Gamma \vdash c\ i \downarrow \tau} \quad (app_c)$$

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

$$\frac{\Delta, \Gamma \vdash i \uparrow \forall t.\tau' \quad \exists \sigma.(\tau = [\sigma/t]\tau')}{\Delta, \Gamma \vdash i[] \downarrow \tau} \quad (instantiate_c)$$

These rules reflect the observations we made in the last section. The trickiest one is for the checkable polymorphic instantiation ($instantiate_c$). Here, we require the existence of a type σ which "fills in the holes" in τ' to get τ. Algorithmically, we can do this by structurally comparing τ and τ', and verifying that any occurence of the type variable t in τ' is matched up with the same type σ in τ.

2.3 Your Job: 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.

Question 1. Complete the PolyMinML type-checker by implementing the functions `check` and `infer` in the file `typing.sml`. 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 ($\forall t.t \to t$ and $\forall u.u \to u$ are the *same type*).
- You will probably want to make use of the functions in `Types` and `PolyMinML` when implementing `check` and `infer`. However, you only need to modify `typing.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. We have provided a number of test cases in the file `testcases.sml`. Tests `t1, t2,` ... should type-check, and tests `f1, f2,` ... should fail. Test one of these expressions with, for example:

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

- Make sure that you test your code, and since our test cases are not at all exhaustive, give yourself time to develop your own.

Question 2. Extend the type checking algorithm and implementation to handle existential types. Be sure to:

- Extend the syntax to define the checkable and inferable existential terms.
- Define the bi-directional type checking inference rules for existentials.
- Extend the `PolyMinML` type checker with existential types. This will involve extending the files `polyminml.sml` (be sure to extend the abstract syntax of `PolyMinML` as well as the substitution functions defined in this file), `types.sml` and `typing.sml`.
- Suitable test cases that test both well-typed and ill-typed expressions.

Hand in all of the above items.

2.4 Further Reading (OPTIONAL)

Benjamin Pierce and David Turner (See the symposium on principles of programming languages, 1998, also available off of Benjamin Pierce's web page) have considered bi-directional type checking in the context of more general type systems involving subtyping and bounded quantification. Haruo Hosoya and Benjamin Pierce have done more experiments concerning the overall effectiveness of bi-directional type checking (see Benjamin Pierce's web page for a further paper). Yitzie Mandelbaum, myself and Robert Harper has been working on a type checking algorithm for still more sophisticated type system that involves theorem proving and dependent types. If you're interested, we can show you a paper we are working on right now.