Assignment 6:
Subtyping and Bidirectional Typing

15-312: Foundations of Programming Languages
Joshua Dunfield (`joshuad@cs.cmu.edu`)

Out: Thursday, October 24, 2002
Due: Thursday, November 7 (**11:59:59 pm**)

100 points total + (up to) 20 points extra credit

1 Introduction

In this assignment, you will implement a bidirectional typechecker for MinML with $\rightarrow$, $*$, $+$, $\mathbf{1}$, $\mathbf{0}$, $\forall$, $\exists$, and subtyping with base types `int` and `float`.

Note: In the .sml files, most changes from Assignment 4 are indicated like this:

```
(* new asst6 code: *)
...
(* end asst6 code *)
```

2 New in this assignment

- Some things have become obsolete (and have either been removed or left in a semi-supported state). Exceptions and continuations are no longer "officially" supported.
- One can now (and sometimes must!) write type annotations $e : \tau$. The abstract syntax constructor is called `Anno`.
- The lexer supports shell-style comments in `.mml` source files: any line beginning with # is ignored.
- There are now two valid syntaxes for functions, the old syntax `fun f (x:t1):t2 is e end` and a new syntax `fun f(x) is e end`. Note the lack of type annotations. The definition of the abstract syntax constructor `Fun` has been changed; its arguments are now just bindings for f and x and the body. It no longer takes two types.

The old syntax `fun f(x:t1):t2 is e end` is now just syntactic sugar, transformed *by the parser* into `fun f(x) is e end : t1 -> t2`.

- There are floating point numbers, written as in SML. There is a new set of arithmetic operators `+.`, `-.`, `*.`, `~.` for use with floating point numbers. `=` and `<` take either integers or floats.

- There is now a void type with concrete syntax `void`. Of course, there is no way to construct a value of type `void`.

- There are now sum types with concrete syntax τ_1 `+` τ_2. A sum type can be constructed via `inl(e)` or `inr(e)`, and taken apart via a `case`:

```
case e of
  inl(x1) => e1
| inr(x2) => e2
end
```

- There are now polymorphic types. The type syntax is `All t.`τ. Unlike SML, type variables are written the same as ordinary variables; `t`, `x`, etc. are all valid type variables. The construct `Fun t in e end` allows one to create an expression of a polymorphic type. In `minml.sml`, its constructor in the abstract syntax is called `Typefun`.[1] A polymorphic expression e can be instantiated at a type τ by writing `e[`τ`]`. The constructor in `minml.sml` is called `Inst`.

  See `ok/poly1.mml`, `ok/poly2.mml`, and `ok/polyvoid.mml` for examples.

- Because we have type variables, the structure `T` that declared the type `T.typ` has become two structures: `NamedT` in which type variables are stored as strings, and `DBT` in which type variables are stored in deBruijn form. `translate.sml` translates the types as well as the program into deBruijn form. You will work exclusively with programs and types in deBruijn form.

- There are now existential types. The type syntax is `Exists t.`τ. The construct `pack tau in e end` allows one to create something of existential type. In `minml.sml`, its constructor in the abstract syntax is called `Pack`. Client code can open up an existential called `module` with `open module as t with impl in e end`; this binds the type variable standing for the opaque type to `t` and the implementation (the body of the `pack`) to `impl` in `e`. The constructor in `minml.sml` is called `Open`.

  See `ok/exis1.mml` and `bad/exis2.mml` for examples.

[1] Just to make this "clear": The constructor `Fun` corresponds to an expression `fun` .... The constructor `Typefun` corresponds to an expression `Fun`. I apologize for the inconvenience.

$$\frac{}{\tau \leq \tau}\ \text{(Refl)}$$

$$\frac{}{\texttt{int} \leq \texttt{float}}\ \text{(IF)}$$

$$\frac{}{\texttt{void} \leq \tau}\ (0)$$

$$\frac{\sigma_1 \leq \tau_1 \quad \sigma_2 \leq \tau_2}{\sigma_1 * \sigma_2 \leq \tau_1 * \tau_2}\ (*) \qquad \frac{\tau_1 \leq \sigma_1 \quad \sigma_2 \leq \tau_2}{\sigma_1 \to \sigma_2 \leq \tau_1 \to \tau_2}\ (\to) \qquad \frac{\sigma_1 \leq \tau_1 \quad \sigma_2 \leq \tau_2}{\sigma_1 + \sigma_2 \leq \tau_1 + \tau_2}\ (+)$$

$$\frac{}{t \leq t}\ (t) \qquad \frac{\{u/s\}\sigma \leq \{u/t\}\tau \quad u \notin FTV(\sigma) \cup FTV(\tau)}{\forall s.\sigma \leq \forall t.\tau}\ (\forall)$$

$$\frac{\{u/t\}\tau \leq \{u/s\}\sigma \quad u \notin FTV(\sigma) \cup FTV(\tau)}{\exists s.\sigma \leq \exists t.\tau}\ (\exists)$$

Figure 1: Subtyping rules. $FTV(\tau)$ denotes the free type variables of τ.

A sad fact: Unfortunately, I didn't have time to update the evaluator so you could actually run programs with the new features. So you can't run most of the examples.

Task: Subtyping (30 points)

Implement the function `subtyping` in `subtyping.sml` according to the rules given.

Note: This is supposed to be relatively easy.

Task: Bidirectional Typing (70 points)

Implement the bidirectional typing system in Figure 2.

Hints

- I decided to be nice and give you a function `typesubst` (in `typing.sml`) for substituting types for type variables. Use it. If you find a bug in it, please let me know immediately.
- The inference and checking judgments are defined in terms of each other. Think about this before you start coding.
- Our type system is no longer syntax-directed—the subsumption rule can be applied anywhere, so think carefully about when it should be applied.
- If you get stuck, ask me. If your question doesn't give something away, ask on the bboard and Cc it to me.

Task: Continuation Typing (up to 20 points EXTRA CREDIT)

Formulate whatever **subtyping** and **bidirectional typing** rules are correct and

$$\frac{\Gamma \vdash e \downarrow \tau \qquad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash (e : \tau) \uparrow \tau} \text{ (Anno)} \qquad \frac{\Gamma \vdash e \uparrow \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e \downarrow \tau} \text{ (Sub)}$$

$$\frac{\Gamma = \Gamma_1, x{:}\tau, \Gamma_2}{\Gamma \vdash x \uparrow \tau} \text{ (Var)}$$

$$\frac{}{\Gamma \vdash \texttt{num}(\overline{k}) \uparrow \texttt{int}} \text{ (Int)} \qquad \frac{\Gamma \vdash e_i \downarrow \sigma_i \qquad \texttt{o} \text{ has type } \sigma_1 * \cdots * \sigma_n \to \tau}{\Gamma \vdash \texttt{o}(e_1, \ldots, e_n) \uparrow \tau} \text{ (Primop)}$$

$$\frac{}{\Gamma \vdash \texttt{Bool}(\ldots) \uparrow \texttt{bool}} \text{ (Bool)} \qquad \frac{\Gamma \vdash e \downarrow \texttt{bool} \qquad \Gamma \vdash e_1 \downarrow \tau \qquad \Gamma \vdash e_2 \downarrow \tau}{\Gamma \vdash \texttt{if}(e, e_1, e_2) \downarrow \tau} \text{ (If)}$$

$$\frac{}{\Gamma \vdash \texttt{()} \uparrow \texttt{unit}} \text{ (Unit)}$$

$$\frac{\Gamma, f{:}\sigma{\to}\tau, x{:}\sigma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{fun}\, f(x)\, \texttt{is}\, e\, \texttt{end} \downarrow \sigma \to \tau} \text{ (Fun)} \qquad \frac{\Gamma \vdash e_1 \uparrow \sigma \to \tau \qquad \Gamma \vdash e_2 \downarrow \sigma}{\Gamma \vdash e_1 e_2 \uparrow \tau} \text{ (Apply)}$$

$$\frac{\Gamma \vdash e_1 \uparrow \sigma \qquad \Gamma, x{:}\sigma \vdash e_2 \downarrow \tau}{\Gamma \vdash \texttt{let}(e_1, x.e_2) \downarrow \tau} \text{ (Let)}$$

$$\frac{\Gamma \vdash e_1 \downarrow \tau_1 \qquad \Gamma \vdash e_2 \downarrow \tau_2}{\Gamma \vdash (e_1, e_2) \downarrow \tau_1 * \tau_2} \text{ (Pair)} \qquad \frac{\Gamma \vdash e \uparrow \tau_1 * \tau_2}{\Gamma \vdash \texttt{fst}(e) \uparrow \tau_1} \text{ (Fst)} \qquad \frac{\Gamma \vdash e \uparrow \tau_1 * \tau_2}{\Gamma \vdash \texttt{snd}(e) \uparrow \tau_2} \text{ (Snd)}$$

$$\frac{\Gamma \vdash e \uparrow \tau_1 + \tau_2 \qquad \Gamma, x_1{:}\tau_1 \vdash e_1 \downarrow \sigma \qquad \Gamma, x_2{:}\tau_2 \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{case}(e, x_1.e_1, x_2.e_2) \downarrow \sigma} \text{ (Case)}$$

$$\frac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \texttt{inl}(e) \downarrow \tau_1 + \tau_2} \text{ (Inl)} \qquad \frac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{inr}(e) \downarrow \tau_1 + \tau_2} \text{ (Inr)}$$

$$\frac{\Gamma, t \text{ type} \vdash e \downarrow \{t/s\}\sigma}{\Gamma \vdash \texttt{Typefun}(t.e) \downarrow \forall s.\sigma} \text{ (Typefun)} \qquad \frac{\Gamma \vdash e \uparrow \forall t.\sigma \qquad \Gamma \vdash \tau \text{ type}}{\Gamma \vdash \texttt{Inst}(e, \tau) \uparrow \{\tau/t\}\sigma} \text{ (Inst)}$$

$$\frac{\Gamma \vdash \sigma \text{ type} \qquad \Gamma \vdash e \downarrow \{\sigma/t\}\tau}{\Gamma \vdash \texttt{pack}(\sigma, e) \downarrow \exists t.\tau} \text{ (Pack)} \qquad \frac{\Gamma \vdash e \uparrow \exists t.\tau \qquad \Gamma, t \text{ type}, x{:}\tau \vdash e' \downarrow \sigma \qquad \Gamma \vdash \sigma \text{ type}}{\Gamma \vdash \texttt{open}(e, t.x.e') \downarrow \sigma} \text{ (Open)}$$

Figure 2: Bidirectional typing rules

appropriate for the continuation type τ `cont` and the constructs `letcc` and `throw` (see Assignment 4). Then find the appropriate **coercion**(s) for the subtyping rule(s).

Turn in the rules as a file `extra.ps` or `extra.pdf`, or on paper by midnight Thursday in WeH 1313. You do not need to implement anything.

WARNING: Right now, I have no idea if this question is easy or hard. If it turns out to be easy, I won't give more than 10 points even for a 100% correct answer. If it turns out to be tricky, then a complete and correct answer could get 20 points.

Test Cases

You are encouraged to submit test cases to us. We will test everyone's code against a subset of the submitted test cases, in addition to our own. So, even though you will not receive any points specifically for handing in test cases, it's in your interest to send us tests that your code handles correctly. See below for submission instructions.

If you like, use the astounding new support for comments in `.mml` source to describe what your tests do.

3 Hand-in Instructions

Turn in the files `subtyping.sml` and `typing.sml` by copying them to your handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/asst6/`

by 11:59 pm on the due date. Immediately after the deadline, we will run a script to sweep through all the handin directories and copy your files elsewhere. We will also sweep 24, 48, and 72 hours after the deadline, for anyone using late days on this assignment.

WARNING: In the past, I have allowed a few minutes leniency, considering files turned in a few minutes after midnight to be "on time". **I will no longer do so.** If any `.sml` file was not in the directory at midnight, I will consider your assignment late.

WARNING: MAKE SURE YOUR TABS ARE SET TO 8 SPACES, or replace tabs with spaces before submitting. I am sick of reading code with messed-up indentation.

Also, please turn in any test cases you'd like us to use by copying them to your handin directory. To ensure that we notice the files, make sure they have the suffix `.mml`. You are welcome to turn these in as late as you like (though I won't notice any files turned in more than 72 hours late).

For more information on handing in code, refer to

`http://www.cs.cmu.edu/~fp/courses/312/assignments.html`