Assignment 6:
Bi-Directional Typechecking

15-312: Foundations of Programming Languages
Matthew Moore (`mlmlm@cmu.edu`)

Out: Thursday, October 21, 2004
Due: Thursday, November 4, 2004 (11:59 pm)

100 points total

1 Background

In this assignment, you will be writing a more practical method for type checking called Bi-Directional Typechecking. This differs from plain type synthesis in that is takes "modes" into account. We distinguish between when we are "synthesizing" a type and when we are "checking" a type. Thus the name, because when we are checking ($\downarrow$), the type is provided from below, but when we are synthesizing ($\uparrow$), the type comes from above (as output).

In addition we will be extending our type system from the previous programming assignment to include Parametric Polymorphism, Data Abstraction, Recursive types, and Subtyping, which you should be familiar with from lecture. (We have also omitted exceptions)

2 Introduction

For this assignment you will just be implementing the static semantics of MinML. *Do not underestimate this assignment*, we have a sophisticated algorithm for type-checking and an advanced type system!

In the assignment directory you'll find several files with support code; you will need to fill in the missing code in `typing.sml`. The resulting module should ascribe opaquely to the signature specified.

You will rarely, if ever, need to write long or complicated functions to complete this assignment. Therefore, you should strive for elegance. Your solution will be graded primarily on correctness, but if your code does not correctly handle one or more cases, we will inspect your code and attempt to give you some credit for the understanding it reflects. You will also have the opportunity to reuse your solution to this assignment in future assignments. In each of the latter situations, it is to your benefit to write clean, legible code.

Before you begin, you may wish to (re)read the provided code (especially the signatures) to gain an understanding of the setup. All of the necessary SML files are listed in the `sources.cm` file, and you can build the project in SML/NJ by typing `CM.make()`.

3 Changes to MinML syntax

Now, instead of having certain language elements which require type annotations, such as $\texttt{inl}(\tau_2, e)$, we only annotate where necessary. We will do this by adding a new form of syntax: $e : \tau$, which will be the only place where explicit type annotation is necessary. It is not immediately clear that this will allow us to have fewer type annotations throughout our MinML code, but once you see the typing rules for our two new methods, you should try to convince yourself that more annotations can be omitted than are needed in other places.

4 Subtyping

Recall from lecture the following basic rules for subtyping:

$$\frac{}{\tau \sqsubseteq \tau} \qquad \frac{}{\texttt{int} \sqsubseteq \texttt{float}} \qquad \frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 * \tau_2 \sqsubseteq \sigma_1 * \sigma_2} \qquad \frac{\tau_1 \sqsubseteq \sigma_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 + \tau_2 \sqsubseteq \sigma_1 + \sigma_2}$$

$$\frac{\sigma_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \sigma_2}{\tau_1 \to \tau_2 \sqsubseteq \sigma_1 \to \sigma_2} \qquad \frac{\tau \sqsubseteq \sigma}{\exists\alpha.\tau \sqsubseteq \exists\alpha.\sigma} \qquad \frac{\tau \sqsubseteq \sigma}{\forall\alpha.\tau \sqsubseteq \forall\alpha.\sigma} \qquad \frac{\tau \sqsubseteq \sigma}{\mu\alpha.\tau \sqsubseteq \mu\alpha.\sigma}$$

Task: Coercion Subtyping (20 points)

Write a function `subtype` which takes in τ, σ, and produces a coercion (a MinML function of type $\tau \to \sigma$) witnessing: $\tau \sqsubseteq \sigma$ based on the above rules. You may make use of a primitive construct `itof` which takes `int`s to `float`s for the coercion from `int` to `float`. In the event that $\tau \not\sqsubseteq \sigma$ you should raise an exception. As an example, the coercion that should be returned for: $\texttt{int} + \texttt{bool} \sqsubseteq \texttt{float} + \texttt{bool}$ should be (observationally equivalent to):

```
fn x =>
  case x of
    inl(y) => inl(itof(y))
  | inr(z) => inr(z)
```

5 Bi-Directional Typechecking

Bi-Directional Typechecking takes the idea of synthesis to a new level by operating in two directions. In ordinary synthesis, every expression has its type generated from only knowing the expression and the typing environment. In Bi-Directional Typechecking, we have two "modes" of typechecking: synthesizing and checking. In checking, we know what type we expect a given expression to have, and carry that type with us, "checking" that the type is valid for the expression. Synthesis is largely the same as before, although when we know the type a given expression should have before "synthesizing" it, we "check" it instead. Figures 1 and 2 give the rules for Bi-Directional typechecking.

$$\frac{}{\Gamma \vdash \texttt{num}(n) \downarrow \texttt{int}} \qquad \frac{}{\Gamma_1, x \uparrow \tau, \Gamma_2 \vdash x \uparrow \tau} \qquad \frac{}{\Gamma \vdash \texttt{true} \downarrow \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} \downarrow \texttt{bool}}$$

$$\frac{\Gamma \vdash e \downarrow \tau_1}{\Gamma \vdash \texttt{inl}(e) \downarrow \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{inr}(e) \downarrow \tau_1 + \tau_2} \qquad \frac{\Gamma, x \uparrow \tau_1 \vdash e \downarrow \tau_2}{\Gamma \vdash \texttt{fn}(x.e) \downarrow \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e \downarrow \{\mu t.\sigma / t\}\sigma}{\Gamma \vdash \texttt{roll}(e) \downarrow \mu t.\sigma}$$

$$\frac{\Gamma, t\ \texttt{type} \vdash e \downarrow \sigma}{\Gamma \vdash \texttt{Fn}(t, e) \downarrow \forall t.\sigma} \qquad \frac{\Gamma \vdash \tau\ \texttt{type} \quad \Gamma \vdash e \downarrow \{\tau/t\}\sigma}{\Gamma \vdash \texttt{pack}(\tau, e) \downarrow \exists t.\sigma} \qquad \frac{\Gamma \vdash e_1 \downarrow \tau_1 \quad \Gamma \vdash e_2 \downarrow \tau_2}{\Gamma \vdash \texttt{pair}(e_1, e_2) \downarrow \tau_1 * \tau_2} \qquad \frac{}{\Gamma \vdash \texttt{flt}(f) \downarrow \texttt{float}}$$

$$\frac{\Gamma, x \uparrow \tau \vdash e \downarrow \tau}{\Gamma \vdash \texttt{rec}(x.e) \downarrow \tau} \qquad \frac{}{\Gamma \vdash \langle\,\rangle \downarrow 1} \qquad \frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash (e : \tau) \uparrow \tau} \qquad \frac{\Gamma \vdash e \uparrow \tau \quad \tau \sqsubseteq \sigma}{\Gamma \vdash e \downarrow \sigma}$$

Figure 1: Constructors, Subsumption and Annotation

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

$$\frac{\Gamma \vdash e_1 \uparrow \tau \quad \Gamma, x \uparrow \tau \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{let}(e_1, x.e_2) \downarrow \sigma} \qquad \frac{\Gamma \vdash e_1 \uparrow \exists t'.\tau \quad \Gamma, t\ \texttt{type}, x \uparrow \{t/t'\}\tau \vdash e_2 \downarrow \sigma}{\Gamma \vdash \texttt{open}(e_1, t.x.e_2) \downarrow \sigma}$$

$$\frac{\Gamma \vdash e \uparrow \forall t.\sigma \quad \Gamma \vdash \tau\ \texttt{type}}{\Gamma \vdash e[\tau] \uparrow \{\tau/t\}\sigma} \qquad \frac{\Gamma \vdash e_1 \uparrow \texttt{bool} \quad \Gamma \vdash e_2 \downarrow \sigma \quad \Gamma \vdash e_3 \downarrow \sigma}{\Gamma \vdash \texttt{if}(e_1, e_2, e_3) \downarrow \sigma}$$

$$\frac{\Gamma \vdash e \uparrow \mu t.\sigma}{\Gamma \vdash \texttt{unroll}(e) \uparrow \{\mu t.\sigma / t\}\sigma} \qquad \frac{\Gamma \vdash e_1 \downarrow \tau_{o1} \quad \ldots \quad \Gamma \vdash e_n \downarrow \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) \uparrow \tau_o}$$

$$\frac{\Gamma \vdash e \uparrow \tau * \sigma}{\Gamma \vdash \texttt{fst}(e) \uparrow \tau} \qquad \frac{\Gamma \vdash e \uparrow \sigma * \tau}{\Gamma \vdash \texttt{snd}(e) \uparrow \tau}$$

$$\frac{\Gamma \vdash e \uparrow 0}{\Gamma \vdash \texttt{abort}(e) \downarrow \tau} \qquad \frac{\Gamma \vdash t\ \texttt{type} \quad \Gamma, x\ \texttt{type} \vdash \{t/x\}e \downarrow \sigma}{\Gamma \vdash \texttt{lettype}(t, x.e) \downarrow \sigma}$$

Figure 2: Destructors

Task: Elaborating Type Checker (70 points)

Fill in the typechecker module, this entails writing two mutually recursive functions, one for synthesis and one for checking. Notice that since we now have subtyping in our language, in order to properly evaluate a well-typed expression, all the places where subtyping was used in checking the program, we must now insert coercions. So instead of simply checking an expression for type safety, we must return a modified version of the expression that coerces expressions where necessary. During this transformation, you should also remove the occurences of type annotation because its sole purpose was to assist the typechecker. This form of type checking coupled with transformations is also known as *elaboration*. In the case that a program is well-typed, the typechecker should return a modified version with coercions inserted as needed and the annotations removed. If the program is not well-typed, you may simply raise an exception.

Task: Data Abstraction (10 points)

In nat.mml implement an abstract type for natural numbers with signature:
$\exists t.t * (t \to t) * (\forall s.t \to (1 \to s) \to (t \to s) \to s)$

6 Test Cases

Filename	**Expected Type**	Expected Result	**Description**
`sqrt.mml`	`float`	3.0	tests coercion and the sqrt operation
`arith.mml`	`float`	27.247...	tests coercion and some arithmetic operators
`ppoly.mml`	(`int`) * (`bool`)	(3, `true`)	tests parametric polymorphism
`exist.mml`	Exists _ . ((DB[1]) -> (DB[1]))	*depends on coercions*	tests data abstraction

NOTE: It is alright (and actually expected) for the result of evaluation not to typecheck because we have removed all the type annotations

You are encouraged to submit test cases to us. We will test each submission 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.

7 Hand-in Instructions

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

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/P3/`

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.

Turn in non-programming questions as text or postscript files in the handin directory. Or, if you wish, you may turn in answers on paper, due in the instructor's office by 11:59 pm on the due date. **If you are using late days, paper handin is by arrangement only** (send mail and we'll figure something out).

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`.

For more information on handing in code, refer to

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