CS 510 Programming Languages
Assignment 4

Out: Monday, October 6, 2003
Due: Monday, October 20, 2003

Written Work

Read Benjamin Pierce's textbook, chapters 9, 11.

Part A) Typability.

1) Are there any types τ_1 and τ_2 such that the following MinML function $\texttt{fun}\ f(x : \tau_1) : \tau_2 = (f\ x)\ (f\ x) : \tau_2$ type checks? Don't wave your hands or guess: prove it definitively yes or no.

Part B) Record typing.

Most languages (including ML) have some sort of record construct. Records are just like tuples except that instead of having each field be named "1," "2," "3," etc., the programmer gets to choose the names like "foo" and "bar" (usually called *labels*).

In this question, we will develop an extension to MinML with records and several interesting operations for manipulating records. We will use the metavariable l to range over labels (just like we use the metavariable n to range over integers $..., -2, -1, 0, 1, 2, ...$). Labels are just strings of characters(*foo* and *bar* can be labels, for instance).

The syntax of the extensions is specified below.

$$\begin{array}{lcll}
\tau & ::= & ... \mid \{l_1 : \tau_1, ..., l_k : \tau_k\} & \text{type of record with } k \text{ fields} \\
e & ::= & ... \mid \{l_1 : e_1, ..., l_k : e_k\} & \text{create record with } k \text{ fields} \\
 & & \mid \#l\ e & \text{select component named } l \text{ from record } e \\
 & & \mid e_1 \ \texttt{concat}\ e_2 & \text{create record containing the fields of } e_1 \text{ and } e_2 \\
 & & \mid e \ \texttt{without}\ l & \text{create record containing all fields of } e \text{ except } l \\
v & ::= & ... \mid \{l_1 : v_1, ..., l_k : v_k\} & \text{record value with } k \text{ fields}
\end{array}$$

There is one other extra syntactic constraint on the formation of records. Records are not allowed to have any fields defined twice. In other words, $\{l_1 : e_1, ..., l_k : e_k\}$ and $\{l_1 : v_1, ..., l_k : v_k\}$ are syntactically ill formed if the same label appears twice in the sequence $l_1, ..., l_k$.

The single-step operational semantics is specified below. The judgment form is $e \longrightarrow e'$, as it has been for other forms of MinML.

$$\frac{e_{i+1} \longrightarrow e'_{i+1}}{\{l_1 : v_1, ..., l_i : v_i, l_{i+1} : e_{i+1}, ..., l_k : e_k\} \longrightarrow \{l_1 : v_1, ..., l_i : v_i, l_{i+1} : e'_{i+1}, ..., l_k : e_k\}}$$

$$\frac{}{\#l_i\ \{l_1 : v_1, ..., l_i : v_i, ..., l_k : v_k\} \longrightarrow v_i}$$

$$\frac{e_1 \longrightarrow e_1'}{\#l\ e_1 \longrightarrow \#l\ e_1'}$$

$$\frac{(\{l_1, ..., l_k\} \cap \{l_{k+1}, ..., l_n\} = \emptyset)}{\{l_1 : v_1, ..., l_k : v_k\}\ \texttt{concat}\ \{l_{k+1} : v_{k+1}, ..., l_n : v_n\} \longrightarrow \{l_1 : v_1, ..., l_k : v_k, l_{k+1} : v_{k+1}, ..., l_n : v_n\}}$$

(The side condition in the premise of the rule above says that the intersection of the two sets of labels in each record value must be the empty set. In other words, the two records that are joined together cannot have any labels in common. Otherwise, this rule does not apply.)

$$\frac{e_1 \longrightarrow e_1'}{e_1\ \texttt{concat}\ e_2 \longrightarrow e_1'\ \texttt{concat}\ e_2}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1\ \texttt{concat}\ e_2 \longrightarrow v_1\ \texttt{concat}\ e_2'}$$

$$\frac{}{\{l_1 : v_1, ..., l_i : v_i, ..., l_k : v_k\}\ \texttt{without}\ l_i \longrightarrow \{l_1 : v_1, ..., ..., l_k : v_k\}}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ \texttt{without}\ l \longrightarrow e_1'\ \texttt{without}\ l}$$

Answer the following questions:

1. Define the typing rules for this extension to MinML.

2. Prove that these extensions are safe by adapting the proof of MinML safety. In other words, I want you to do the cases of the proofs of all of the necessary lemmas that are relevant to the record extensions. You can assume that the substitution lemma has already been proven. I want to see the:

a) Canonical forms lemma.

b) Inversion of typing lemma.

c) Progress lemma.

d) Preservation lemma.

and any other necessary components.

3. In MinML with records and functions, if $\vdash e : \tau$ and $e \longrightarrow^* e'$ then $FV(e') = \emptyset$. Prove it.

4. On pages 42-43, Pierce describes what is known as the "big-step" operational semantics for the language of arithmetic expressions. Extend the big-step semantics given on these pages with rules for MinML functions and records.

5. State and prove a theorem expressing that typing for MinML with functions and records is preserved by the big-step semantics you have defined. (ie: state and prove the Preservation Theorem as it relates to the big-step semantics rather than the single-step semantics). You may use any lemmas you have proven earlier.

6. As I was making up this assignment, I got a phone call from Joe Hacker. He doesn't like the record calculus as I have defined it. He wants the record calculus to be more flexible than it currently is. In particular, he wants labels to be expressions that can be computed with (returned from functions, stored in data structures, etc.) just like any other sort of expression. Joe Hacker suggests that we extend the syntax of our calculus as follows:

$$\begin{array}{lcl} e & ::= & ... \mid l \mid \#e_2\ e_1 \mid e_1\ \texttt{without}\ e_2 \\ v & ::= & ... \mid l \end{array}$$

Explain in detail what impact this change has on the language and its safety properties. More specifically, provide (reasonable) typing rules (you can also extend the set of allowed types) and operational semantics (small step) for the modified constructs. Explain why your simple (but reasonable) typing rules are unsound and why it is difficult to provide simple typing rules like the ones given class that guarantee that programs in the modified language do not get stuck (or result in checked errors).

Programming Work

For the programming part of this assignment you will implement a typechecker for MinML. On the course assignment web site, you'll find several files with support code; you will only need to fill in the missing code in `typing.sml` to complete the assignment.

Before you begin, you may wish to read through 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()`.

Parsing

The file `parse.sml` contains the beginnings of a parser for MinML. The `parse` function turns a `Lex.token Stream.stream` into a `MinML.exp Stream.stream` by consuming programs (which are expressions followed by a semicolon). For simplicity, we will not attempt error recovery; if the parser encounters an error it will just raise the exception ParseError with an informative message.

Here is the concrete syntax we will use. To simplify your job further, this syntax uses no infix notation.

```
Op ::= PLUS | MINUS | TIMES | EQUALS | NEGATE

Type ::= INT | BOOL | ARROW Type Type

ExpSeq ::= Exp | Exp COMMA ExpSeq

Var ::= VAR(s)

Exp ::= LPAREN Exp RPAREN
      | NUMBER(n)
      | Op LPAREN ExpSeq RPAREN
      | Exp Exp
      | TRUE
      | FALSE
      | IF Exp THEN Exp ELSE Exp FI
      | FUN Var LPAREN Var COLON Type RPAREN COLON Type IS Exp END

Program ::= Exp SEMICOLON
```

This syntax should be mostly self-explanatory. Note in particular that primitive operations are prefix (followed by one or more expressions separated by commas and surrounded by parentheses). Types are also written in prefix notation. Application is written by juxtaposition, as in ML. Here are some examples along with their translation into MinML abstract syntax (MinML.exp).

Concrete Syntax	**Abstract Syntax**
`true`	`Bool(true)`
`1`	`Int(1)`
`if true then 4 else 5 fi`	`If(Bool(true), Int(4), Int(5))`
`f 3`	`Apply(Var(''f''), Int(3))`
`+(1, 2)`	`Primop(Plus, Int(1), Int(2))`
`fun f(g :  -> int -> bool int) :  bool is true end`	`Fun(''f'', ''g'', ARROW(INT,ARROW(bool,int)), BOOL, Bool(true))`

Typechecker

We ask you to implement a typechecker for MinML. The static semantics for MinML, given in Figure 1, assures that every expression has at most one type. Therefore, your typechecker will return the unique type for an expression if it is well-typed, or raise the exception `TypeError` otherwise.

Recall that the specification of MinML uses a *typing judgment* to classify MinML expressions as ill- or well-typed. The typing judgment is defined inductively by the inference rules. Therefore, in order to decide whether a given expression has a type, we need to search for a derivation using the typing rules. However, a moment of thought realizes that if we could classify an expression as ill- or well-typed through a typing judgment, then we could also retrieve its type easily, because the derivation itself would tell exactly how to determine the type of the expression. Therefore we see that deciding the type of an expression is no harder than simply classifying the expression as ill- or well-typed.

In general, we cannot assume that an expression matches only one typing rule and thus, the search strategy for a derivation can be *non-deterministic*. Fortunately, the search strategy for MinML is *syntax directed*: the form of expression we are typing determines uniquely which rule to apply. Therefore, if the typechecker finds that no rule can be applied to an expression, it knows that the expression is ill-typed and can raise an exception immediately without backtracking. Your code will probably have one function clause for each clause of the datatype `MinML.exp`.

We provide in `minml.sml` a function `typeOfPrimop` which returns the domain and range types for a primop. Your typechecker should make use of this function, but should not rely on the fact that our primops currently have a maximum of two arguments (we should be able to extend MinML with additional primops without modifying your type checker).

Complete the code in `typing.sml` to produce a structure `Typing :> TYPING` which implements the behavior specified. You should not modify any other files.

This file contains some code to get you started; we recommend using it. The majority of your work will be done on the function `typing`, with type `(string, typ) env * exp -> typ` and specification as follows:

Given an environment Γ and an expression e, `typing` should return τ, the type of e under Γ, if τ exists. If e is ill-typed in Γ then `typing` should raise the exception `TypeError`.

You can test your typechecker by running:
`Top.loop_type ()` or `Top.file_type "test_file"`.
If you're interested, you can see an implementation of the MinML dynamic semantics in the file `eval.sml`.
We've provided a few test MinML programs. Once your assignment is complete, these may be run in the following manner:

```
Top.file_eval "test_file.mml";
```

Filename	**Expected Result**	**Description**
`if.mml`	`3 : int`	Simple test of `if`
`fun.mml`	`fun ident(x : int) : int is x end : -> int int`	Simple test of `fun`
`factorial.mml`	`120 : int`	The factorial function
`self.mml`	*ill-typed*	Ill-typed function
`hof.mml`	`7 : int`	Simulates pairs using functions

EXTRA CREDIT: Extend this implementation with records.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ VarTyp$$

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ NumTyp$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ TrueTyp$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ FalseTyp$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ AppTyp$$

$$\frac{\Gamma \vdash e_c : \texttt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\, e_c\, \texttt{then}\, e_t\, \texttt{else}\, e_f\, \texttt{fi} : \tau}\ IfTyp$$

$$\frac{\Gamma \vdash e_1 : \tau_{o1} \quad ... \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) : \tau_o}\ OpTyp$$

$$\frac{\Gamma[f : \tau_1 \to \tau_2][x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f(x : \tau_1) : \tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2}\ FunTyp$$

Figure 1: Static Semantics for MinML