CS 15-312 Assignment #2
Extending MinMLwith Products, Sums and Recursive Types

Assigned: Friday, February 11
Due: Friday, February 25, 6 PM

1 Introduction

In this assignment we extend MinMLwith Products, Sums and Recursive Types; you are asked to:

1. Prove preservation and progress for the overall extended language;
   - Preservation: assume $e \mapsto e'$; then if $e : \tau$, then $e : \tau'$.
   - Progress: if $e : \tau$, then either e is a value or there exists e' such that $e \mapsto e'$.
2. Implement static and dynamic semantics for it.

Note: recursive types are **extra** credit. Here's the breakdown:

- Products and Sums: preservation 30 pts, progress 30 pts, implementation 40 pts.
- Recursive types: preservation 10 pts, progress 10 pts, implementation 30 pts.

2 Product Types

Think of products as the type of ML tuples. However, in our extension to MinML, we will only add binary tuples (pairs), though they can be nested. For pair destructor, i.e. expression that takes apart a value of type product, we introduce a **bind** expression for pairs that uses a limited form of pattern matching. As a special case (the 0-ary product) we introduce the type **1** which has only one value, namely $\langle\rangle$.

$$\begin{array}{rcll} \textit{Types} & \tau & ::= & \ldots \mid \tau_1 * \tau_2 \mid \mathbf{1} \\ \textit{Expressions} & e & ::= & \ldots \mid \langle e_1, e_2 \rangle \mid \mathbf{bind}\, p \;\mathbf{to}\; e_1 \;\mathbf{in}\; e_2 \mid \langle\rangle \\ \textit{Patterns} & p & ::= & \_ \mid x : \tau \mid \langle p_1, p_2 \rangle \mid \langle\rangle \\ \textit{Values} & v & ::= & \ldots \mid \langle v_1, v_2 \rangle \mid \langle\rangle \end{array}$$

We have four kinds of patterns: wild card patterns which can take any type, variable patterns which can take only the specified type, empty patterns of type **1** and pair patterns which take the product of the types of their constituent patterns. We need to define a new typing judgment on patterns of the form $\Gamma \vdash p : \tau \hookrightarrow \Gamma'$. This judgment will essentially do pattern matching on p to determine whether it can be given type τ, and if so, add the variables contained in p to the context Γ with their respective types, producing Γ'.

$$\frac{}{\Gamma \vdash \_ : \tau \hookrightarrow \Gamma}[WildPat] \qquad \frac{}{\Gamma \vdash (x{:}\tau) : \tau \hookrightarrow \Gamma[x{:}\tau]}[VarPat]$$

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1} \hookrightarrow \Gamma}[EmptyPat] \qquad \frac{\Gamma \vdash p_1 : \tau_1 \hookrightarrow \Gamma' \quad \Gamma' \vdash p_2 : \tau_2 \hookrightarrow \Gamma''}{\Gamma \vdash \langle p_1, p_2\rangle : \tau_1 \times \tau_2 \hookrightarrow \Gamma''}[PairPat]$$

Typing rules:

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}}[UnitTyp] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2}[PairTyp]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash p : \tau_1 \hookrightarrow \Gamma' \quad \Gamma' \vdash e_2 : \tau}{\Gamma \vdash \mathbf{bind}\ p\ \mathbf{to}\ e_1\ \mathbf{in}\ e_2 : \tau}[BindTyp]$$

The evaluation rules are as expected. The only ones that are a little tricky are those involving `bind` because we need to do pattern matching on the pattern in the `bind` and, if it is a pair pattern, break the expression into two nested `bind`'s. Aside from that, left-to-right evaluation prevails as usual.

Evaluation Rules:

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}[PairLeft] \qquad \frac{e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}[PairRight]$$

$$\frac{e_1 \mapsto e_1'}{\mathbf{bind}\ p\ \mathbf{to}\ e_1\ \mathbf{in}\ e_2 \mapsto \mathbf{bind}\ p\ \mathbf{to}\ e_1'\ \mathbf{in}\ e_2}[BindRHS] \qquad \frac{}{\mathbf{bind}\ \langle\rangle\ \mathbf{to}\ v\ \mathbf{in}\ e \mapsto e}[BindEmpty]$$

$$\frac{}{\mathbf{bind}\ \_\ \mathbf{to}\ v\ \mathbf{in}\ e \mapsto e}[BindWild] \qquad \frac{}{\mathbf{bind}\ x : \tau\ \mathbf{to}\ v\ \mathbf{in}\ e \mapsto [v/x]e}[BindVar]$$

$$\frac{}{\mathbf{bind}\ \langle p_1, p_2\rangle\ \mathbf{to}\ \langle v_1, v_2\rangle\ \mathbf{in}\ e \mapsto \mathbf{bind}\ p_1\ \mathbf{to}\ v_1\ \mathbf{in}\ (\mathbf{bind}\ p_2\ \mathbf{to}\ v_2\ \mathbf{in}\ e)}[BindPair]$$

3 Sum Types

Think of *sum* types as (non-recursive) ML datatype; for now we will restrict only to *binary datatypes* with fixed constructors $\mathbf{inl}^{\tau_1+\tau_2} e$ and $\mathbf{inr}^{\tau_1+\tau_2} e$ (pronounced "inleft" and "inright") as the datatype constructors. We annotate those constructors with the type they are injecting in to retain uniqueness of typing. For type destructor, we use a **case** expression for sums.

$$\begin{array}{rrcl} \textit{Types} & \tau & ::= & \ldots \mid \tau_1 + \tau_2 \\ \textit{Expressions} & e & ::= & \ldots \mid \mathbf{inl}^{\tau_1+\tau_2} e \mid \mathbf{inr}^{\tau_1+\tau_2} e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}\ x_1 : \tau_1 \Rightarrow e_1 \mid \mathbf{inr}\ x_2 : \tau_2 \Rightarrow e_2 \\ \textit{Values} & v & ::= & \ldots \mid \mathbf{inl}^{\tau_1+\tau_2} v \mid \mathbf{inr}^{\tau_1+\tau_2} v \end{array}$$

For example:

$$\begin{array}{rcl} bool &=& \mathbf{1}+\mathbf{1} \\ True &=& \mathbf{inl}^{\mathbf{1}+\mathbf{1}}\langle\rangle \\ False &=& \mathbf{inr}^{\mathbf{1}+\mathbf{1}}\langle\rangle \\ If\ e\ Then\ e_1\ Else\ e_1 &=& \mathbf{case}\ e\ \mathbf{of\ inl}\ \langle\rangle : \mathbf{1}\ \Rightarrow\ e_1 \mid \mathbf{inr}\ \langle\rangle : \mathbf{1}\ \Rightarrow\ e_2 \end{array}$$

Type rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{inl}^{\tau_1+\tau_2} e : \tau_1 + \tau_2}[InlType] \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{inr}^{\tau_1+\tau_2} e : \tau_1 + \tau_2}[InrType]$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma[x_1{:}\tau_1] \vdash e_1 : \tau_1 \qquad \Gamma[x_2{:}\tau_2] \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of\ inl}\ x_1 : \tau_1 \Rightarrow e_1 \mid \mathbf{inr}\ x_2 : \tau_2 \Rightarrow e_2 : \tau_1 + \tau_2}[CaseTyp]$$

Evaluation rules:

$$\frac{e \mapsto e'}{\mathbf{inl}^{\tau_1+\tau_2} e \mapsto \mathbf{inl}^{\tau_1+\tau_2} e'}[Inl] \qquad \frac{e \mapsto e'}{\mathbf{inr}^{\tau_1+\tau_2} e \mapsto \mathbf{inr}^{\tau_1+\tau_2} e'}[InR]$$

$$\frac{}{\mathbf{case\ inl}^{\tau_2} v\ \mathbf{of\ inl}\ x_1 : \tau_1\ \Rightarrow\ e_1 \mid \mathbf{inr}\ x_2 : \tau_2\ \Rightarrow\ e_2 \mapsto [v/x_1]e_1}[CaseLeft]$$

$$\frac{}{\mathbf{case\ inr}^{\tau_1} v\ \mathbf{of\ inl}\ x_1 : \tau_1\ \Rightarrow\ e_1 \mid \mathbf{inr}\ x_2 : \tau_2\ \Rightarrow\ e_2 \mapsto [v/x_2]e_2}[CaseRight]$$

$$\frac{e \mapsto e'}{\begin{array}{c}\mathbf{case}\ e\ \mathbf{of\ inl}\ x_1 : \tau_1\ \Rightarrow\ e_1 \mid \mathbf{inr}\ x_2 : \tau_2\ \Rightarrow\ e_2 \mapsto \\ \mathbf{case}\ e'\ \mathbf{of\ inl}\ x_1 : \tau_1\ \Rightarrow\ e_1 \mid \mathbf{inr}\ x_2 : \tau_2\ \Rightarrow\ e_2\end{array}}[CaseTest]$$

4 Recursive Types

Extra Credit. We have seen as *sum* types can be used to define finitary data types, as booleans, but we lack the ability to introduce types as integers which are typically infinite. One way is to introduce *recursive* types. Recall that $\mathbf{1}+\mathbf{1}$ is isomorphic to the type of the boolean; similarly a finite type with n values is isomorphic to:

$$\underbrace{\mathbf{1}+\ldots+\mathbf{1}}_{n \text{ times}}$$

By analogy the natural numbers would be an infinite sum: this can be expresses by the recursive type $\mu\alpha.\,\mathbf{1}+\alpha$.

Recursive types are built via type variables α and the μ operator; they are equipped with a constructor $\mathbf{fold}^{\mu\alpha.\,\tau}$ and a destructor $\mathbf{unfold}^{\mu\alpha.\,\tau}$. For example:

$$
\begin{array}{rcl}
Nat & = & \mu\alpha.\,\mathbf{1} + \alpha \\
Zero & = & \mathbf{fold}^{Nat}(\mathbf{inl}^{\mathbf{1}+Nat}\langle\rangle) \\
One & = & \mathbf{fold}^{Nat}(\mathbf{fold}^{Nat}(\mathbf{inr}^{\mathbf{1}+Nat)}(\mathbf{inl}^{\mathbf{1}+Nat}\langle\rangle))) \\
Succ & = & \mathbf{fun}\ succ(x : Nat) : Nat\ \mathbf{is}\ (\mathbf{fold}^{Nat}(\mathbf{inr}^{\mathbf{1}+Nat}x))\ \mathbf{end} \\
Pred & = & \mathbf{fun}\ pr(x : Nat) : Nat\ \mathbf{is}\ \mathbf{case}\ \mathbf{unfold}^{nat}x\ \mathbf{of}\ Zero \Rightarrow Zero \mid Succ(y) \Rightarrow y\ \mathbf{end}
\end{array}
$$

You can think of **fold** as a map (a 'predecessor' pointer) $Nat \Leftarrow \mathbf{1} + Nat$, while $\mathbf{unfold}^{nat}$ is its inverse $Nat \Rightarrow \mathbf{1} + Nat$.

$$
\begin{array}{rrcl}
\textit{Types} & \tau & ::= & \ldots \mid \alpha \mid \mu\alpha.\,\tau \\
\textit{Expressions} & e & ::= & \ldots \mid \mathbf{fold}^{\mu\alpha.\,\tau}e \mid \mathbf{unfold}^{\mu\alpha.\,\tau}e \\
\textit{Values} & v & ::= & \ldots \mid \mathbf{fold}^{\mu\alpha.\,\tau}v
\end{array}
$$

Typing rules:

$$
\frac{\Gamma \vdash e : [\mu\alpha.\,\tau/\alpha]\tau}{\Gamma \vdash \mathbf{fold}^{\mu\alpha.\,\tau}e : \mu\alpha.\,\tau}\,[FoldTyp] \qquad \frac{\Gamma \vdash e : \mu\alpha.\,\tau}{\Gamma \vdash \mathbf{unfold}^{\mu\alpha.\,\tau}e : [\mu\alpha.\,\tau/\alpha]\tau}\,[UnfoldTyp]
$$

Evaluation rules:

$$
\frac{}{\mathbf{unfold}^{\mu\alpha.\,\tau}(\mathbf{fold}^{\mu\alpha.\,\tau}e) \mapsto e}\,[FoldBeta]
$$

$$
\frac{e \mapsto e'}{\mathbf{fold}^{\mu\alpha.\,\tau}e \mapsto e'}\,[FoldTest] \qquad \frac{e \mapsto e'}{\mathbf{unfold}^{\mu\alpha.\,\tau}e \mapsto e'}\,[UnfoldTest]
$$

Another example is lists of natural numbers:

$$
\begin{array}{rcl}
List & = & \mu\alpha.\,\mathbf{1} + (nat * \alpha) \\
Nil & = & \mathbf{fold}^{List}(\mathbf{inl}^{\mathbf{1}+List}\langle\rangle) \\
Cons & = & \mathbf{fun}\ cons(x : nat * List) : List\ \mathbf{is}\ (\mathbf{fold}^{List}(\mathbf{inr}^{\mathbf{1}+List}x))\ \mathbf{end}
\end{array}
$$

Thus $Cons$ takes a number and a list and return the list with the number as its head. To check your understanding, give a typing derivation for $Cons(\langle Zero, \langle One, Nil\rangle\rangle)$. Next define a function `length` with tale a list and return its length via **case** and $\mathbf{unfold}^{\mu\alpha.\,\tau}$.

5 Implementation

What you hand in: files `typed.sml`, `eval.sml` and `parse.sml`.

5.1 Parsing

The parser is already provided in `parse.sml`; to check your understanding, you are asked to add rules for the empty pair and only if you choose so for recursive types.

5.2 Typing

File `typing-sig.sml` contains the signature, while `typing.sml` will contain the structure. You are given a `datatype pat =...`. You need to define a function for pattern matching: `match : env*pat*typ -> env`

5.3 Evaluation

As in the previous assignment.

5.4 Recursive types–extra credit

You also need to update the parser to allow recursive types and related constructor. You'll need a way to create fresh type variable:

```
(* creates a fresh type variable *)
 val fresh_tyvar : unit -> tyvar
 (* resets the type variable generator *)
 val reset_tyvar : unit -> unit
 (* converts a type variable to a string *)
 val tyvar2string : tyvar -> string
```

An implementation is in `minml.sml`. You also need to provide a substitution function for *types*, say `tsubst sub*typ -> typ`.