Assignment 4: Adding exceptions and continuations to MinML

15-312: Foundations of Programming Languages

Kevin Watkins ⟨kw@cmu.edu⟩

Out: Thursday, February 17, 2005
Due: Thursday, March 3, 2005 (11:59 p.m.)

100 points total

In this programming assignment, your task will be to extend the MinML type-checker and evaluator of Assignment 2 to work with sum types, product types, exceptions and continuations. The code you'll start with is mostly the starter code for Assignment 2, except that syntax, parsing and printing for all the new features have been added. Your job will be to start with either your solution or the model solution to Assignment 2, and extend the type checker and evaluator for all the new features. (This time, I've done the de Bruijn translator for you.)

As before, you can find the starter code in

`/afs/andrew/scs/cs/15-312/handout/4/`,

and you'll be handing your solution in by copying it to

`/afs/andrew/scs/cs/15-312/handin/`*you*`/4/`.

The files you will be handing in are:

`typing.sml`	`1.mml`
`eval.sml`	`2.mml`
`fail.mml`	`3.mml`

If you don't remember all the advice regarding submitting your solution, grading criteria for programming assignments, and all that jazz, now would be a good time to go back to the problem statement for Assignment 2 and refresh your memory.

The C machine. One thing that's changed since Assignment 2 is that we're using the C machine rather than the M machine. So our `step` function is now going to take a state and return another state, rather than taking an expression and returning another expression.

So how should we represent states in SML? It seems reasonable to have a datatype with a separate constructor for each kind of state, so we'd end up with something

like

$$\begin{array}{ll}
\texttt{datatype state} = & \\
\quad \texttt{Eval of stack} \times \texttt{exp} & \ulcorner k > e \urcorner = \texttt{Eval}(k, e) \\
\quad \mid \texttt{Return of stack} \times \texttt{exp} & \ulcorner k < v \urcorner = \texttt{Return}(k, v) \\
\quad \mid \texttt{Fail of stack} & \ulcorner k \ll \textbf{fail} \urcorner = \texttt{Fail}(k)
\end{array}$$

where on the right I've written the *representation function* $\ulcorner - \urcorner$ showing how each state of the C machine is represented in SML.

Now we have to figure out how to represent stacks. Probably the most straightforward solution would be to have a separate datatype constructor for each kind of stack frame. But there are so many kinds of frames that it would be pretty tedious to list them all. And the different kinds of frames all seem to have pretty much the same structure; they all represent some kind of postponed work that has to be done to finish evaluating an expression. It would be great if we could represent the postponed work as an SML function, and just call the function to do whatever the stack frame is supposed to do.

One way to think about stacks is to consider what kinds of things we'll be doing when we need to look at the stack. In fact, the C machine only looks at the stack when it's trying to take a step in one of the situations $k < v$ or $k \ll \textbf{fail}$. If we know what to do in both of those situations, we're set.

This suggests that our representation for a stack k should be a pair of SML functions (f_1, f_2). The function f_1 says what to do when we're returning a value $k < v$ to the stack k. In that case, we'll call $f_1\ v$ and step to whatever state that is. The other function f_2 says what to do when we're returning failure $k \ll \textbf{fail}$ to the stack k. In that case, we'll just call $f_2\ ()$. Finally, if we want to print states of the machine, it's impossible to print the SML functions f_1 and f_2, but if we store the number of frames with the stack, at least we can see how many there are. With this in mind, we set

$$\texttt{datatype stack} = \texttt{Stack of int} \times (\texttt{exp} \to \texttt{state}) \times (\texttt{unit} \to \texttt{state}).$$

(Actually, this is mutually recursive with `state` and `exp`, so the SML syntax makes us declare them all at the same time.)

The disadvantage of this representation is that we have no way of printing out a stack so that we can see what exactly it looks like at each step of the C machine evaluation. But the convenience of using SML functions for the stack seems to be worth more than the ability to print them out.

These datatypes have been included in the starter code for this assignment. The M machine evaluator from Assignment 2 has been converted to a C machine evaluator based on them. You should look at the supplied code in `eval.sml` now, and make sure you understand how it works. In particular, you need to fully understand the functions `eval`, `ret`, `fail`, `frame`, and `try`. Only `eval`, `ret`, and `frame` are used in the starter code you've been given, but I provided `fail` and `try` in the hope that they will be useful to you when implementing exceptions.

Syntax and parsing. Once again, the parser for the language we're dealing with has been provided for you. Rather than include the grammar for the ASCII syntax here, I'll just mention that it's in the file `parse.sml`.

Note that the syntax has changed slightly from what we used in Assignment 2. I got fed up with the **end**s everywhere, so now they're not necessary. For example,

$$\textbf{fun}\ f(x : \textbf{bool}) : \textbf{int}\ \textbf{is}\ \textbf{if}\ x\ \textbf{then}\ 1\ \textbf{else}\ 2$$

is now legal MinML syntax. A construct like **fun** or **if** extends as far to the right as possible, consistent with the pairing of parentheses and so forth.

Many constructs in the extended language we'll be working with need extra type information to ensure that types are unique. The convention in such cases is that the type of the whole expression is written in brackets after the keyword starting the expression. For example, in $\textbf{letcc}[\tau]\ x\ \textbf{in}\ e$, the type of the whole expression is τ.

I also added comment syntax to the lexical analyzer; any line starting with the character '`%`' is ignored.

§ 1. Product and sum types. The first extensions to the language from Assignment 2 are product and sum types. These have the syntax

$$\tau ::= \ldots \mid 1 \mid \tau_1 \times \tau_2 \mid 0 \mid \tau_1 + \tau_2$$

The corresponding expressions are

$$\begin{aligned} e \quad ::= \quad & \ldots \mid () \mid e_1, e_2 \mid e.1 \mid e.2 \\ & \mid \textbf{abort}[\tau]\, e \mid \textbf{in}_1[\tau]\, e \mid \textbf{in}_2[\tau]\, e \mid \textbf{case}\ e\ \textbf{of}\ \textbf{in}_1\ x_1 \Rightarrow e_1 \mid \textbf{in}_2\ x_2 \Rightarrow e_2 \end{aligned}$$

First, the products. They have static semantics given by the following typing rules:

$$\frac{}{\Gamma \vdash () : 1}\ \mathit{UnitTyp} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_1 \times \tau_2}\ \mathit{PairTyp}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.1 : \tau_1}\ \mathit{FstTyp} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e.2 : \tau_2}\ \mathit{SndTyp}$$

We only treat binary and zero-ary products. The unit type, written 1, is a zero-ary product. Parentheses are optional around pairs e_1, e_2; this means you can use the left-associativity of ',' and '$\times$' together to write things like $1, 2, 3 : \textbf{int} \times \textbf{int} \times \textbf{int}$.

Now, the sums. They have static semantics given by the following typing rules:

$$\frac{\Gamma \vdash e : 0}{\Gamma \vdash \textbf{abort}[\tau]\, e : \tau}\ \mathit{AbortTyp}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, (x_1 : \tau_1) \vdash e_1 : \tau \qquad \Gamma, (x_2 : \tau_2) \vdash e_2 : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \textbf{in}_1\ x_1 \Rightarrow e_1 \mid \textbf{in}_2\ x_2 \Rightarrow e_2 : \tau}\ \mathit{CaseTyp}^{1}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{in}_1[\tau_1 + \tau_2]\, e : \tau_1 + \tau_2}\ \mathit{InlTyp} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{in}_2[\tau_1 + \tau_2]\, e : \tau_1 + \tau_2}\ \mathit{InrTyp}$$

[1] $x_1, x_2 \notin \operatorname{dom} \Gamma$

The void type, written 0, is a zero-ary sum. This can come in handy when we're programming with complicated control structures—continuations or what have you—because we can express the invariant that certain code never returns by giving it a type involving 0. The only expression associated with the void type is **abort**, which

is just like **case** except that it has zero arms instead of two.

Now, for the dynamic semantics. The values for products and sums are

$$v ::= \ldots \mid () \mid v_1, v_2 \mid \mathbf{in}_1[\tau]\, v \mid \mathbf{in}_2[\tau]\, v$$

The C machine rules for products and sums are as follows:

$$k > () \mapsto k < ()$$

$$k > e_1, e_2 \mapsto k \triangleright \Box, e_2 > e_1$$
$$k \triangleright \Box, e_2 < v_1 \mapsto k \triangleright v_1, \Box > e_2$$
$$k \triangleright v_1, \Box < v_2 \mapsto k < v_1, v_2$$

$$k > e.1 \mapsto k \triangleright \Box.1 > e$$
$$k \triangleright \Box.1 < v_1, v_2 \mapsto k < v_1$$
$$k > e.2 \mapsto k \triangleright \Box.2 > e$$
$$k \triangleright \Box.2 < v_1, v_2 \mapsto k < v_2$$

$$k > \mathbf{abort}[\tau]\, e \mapsto k \triangleright \mathbf{abort}[\tau]\, \Box > e$$

$$k > \mathbf{in}_1[\tau]\, e \mapsto k \triangleright \mathbf{in}_1[\tau]\, \Box > e$$
$$k \triangleright \mathbf{in}_1[\tau]\, \Box < v \mapsto k < \mathbf{in}_1[\tau]\, v$$
$$k > \mathbf{in}_2[\tau]\, e \mapsto k \triangleright \mathbf{in}_2[\tau]\, \Box > e$$
$$k \triangleright \mathbf{in}_2[\tau]\, \Box < v \mapsto k < \mathbf{in}_2[\tau]\, v$$

$$k > \mathbf{case}\ e\ \mathbf{of}\ \mathbf{in}_1\, x_1 \Rightarrow e_1 \mid \mathbf{in}_2\, x_2 \Rightarrow e_2 \mapsto k \triangleright \mathbf{case}\ \Box\ \mathbf{of}\ \mathbf{in}_1\, x_1 \Rightarrow e_1 \mid \mathbf{in}_2\, x_2 \Rightarrow e_2 > e$$
$$k \triangleright \mathbf{case}\ \Box\ \mathbf{of}\ \mathbf{in}_1\, x_1 \Rightarrow e_1 \mid \mathbf{in}_2\, x_2 \Rightarrow e_2 < \mathbf{in}_1[\tau]\, v \mapsto k > \{v/x_1\}e_1$$
$$k \triangleright \mathbf{case}\ \Box\ \mathbf{of}\ \mathbf{in}_1\, x_1 \Rightarrow e_1 \mid \mathbf{in}_2\, x_2 \Rightarrow e_2 < \mathbf{in}_2[\tau]\, v \mapsto k > \{v/x_2\}e_2$$

Note that the first rule for **abort** is analogous to the first rule for **case**, but whereas **case** has two more rules, **abort** has zero. Funny how that works out.

Q 1.1. [45 pts] Following the rules above, extend the typing function in `typing.sml` and the `step` function in `eval.sml` to handle products and sums (including the unit and void constructs).

Note that the C machine implemented in `eval.sml` uses SML functions to represent the stack, rather than explicit syntax. Still, the syntax for frames in the rules above should suggest to you what SML functions to use to implement them.

As usual, the `step` function must satisfy the following specification:

> If s is a closed C machine state, and $\ulcorner s \urcorner$ is its SML representation as a `state`, and $s \mapsto s'$, then `step`$(\ulcorner s \urcorner) = \ulcorner s' \urcorner$, where $\ulcorner s' \urcorner$ is the SML representation of the C machine state s'.
>
> If s is a closed C machine state, and $\ulcorner s \urcorner$ is its SML representation, and s cannot take a C machine step, then `step`$(\ulcorner s \urcorner)$ must raise exception `NoStep`.

Note that the specification says nothing about the states involved being well

typed.

§ 2. Exceptions. The next extension to the language introduces exceptions and exception handling. As in lecture, the exceptions you'll be implementing don't carry any values along with them, unlike SML exceptions.

The syntax for exceptions is as follows:

$$e ::= \ldots \mid \textbf{try}\ e_1\ \textbf{ow}\ e_2 \mid \textbf{fail}[\tau]$$

No new types or values are introduced. The static semantics is based on the rules

$$\frac{}{\Gamma \vdash \textbf{fail}[\tau] : \tau}\ \mathit{FailTyp} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{try}\ e_1\ \textbf{ow}\ e_2 : \tau}\ \mathit{TryTyp}$$

and the C machine dynamic semantics is based on the rules

$$\begin{aligned}
&k > \textbf{try}\ e_1\ \textbf{ow}\ e_2 \mapsto k \triangleright \textbf{try}\ \Box\ \textbf{ow}\ e_2 > e_1\\
&k \triangleright \textbf{try}\ \Box\ \textbf{ow}\ e_2 < v \mapsto k < v\\
&k \triangleright \textbf{try}\ \Box\ \textbf{ow}\ e_2 \ll \textbf{fail} \mapsto k > e_2\\
&k \triangleright f \ll \textbf{fail} \mapsto k \ll \textbf{fail} \quad \text{if } f \neq \textbf{try}\ \Box\ \textbf{ow}\ e_2\\
&k > \textbf{fail}[\tau] \mapsto k \ll \textbf{fail}
\end{aligned}$$

Q 2.1. [20 pts] As before, extend `typing.sml` and `eval.sml` to treat the new constructs.

§ 3. First-class continuations. The last extension to the language adds first-class continuations in the style we saw in lecture. The new types are

$$\tau ::= \ldots \mid \tau\ \textbf{cont}$$

while the new expressions are

$$e ::= \ldots \mid \textbf{letcc}[\tau]\ x\ \textbf{in}\ e \mid \textbf{throw}[\tau]\ e_1\ \textbf{to}\ e_2 \mid \textbf{cont}(k)$$

The static semantics is given by the rules

$$\frac{\Gamma, (x : \tau\ \textbf{cont}) \vdash e : \tau}{\Gamma \vdash \textbf{letcc}[\tau]\ x\ \textbf{in}\ e : \tau}\ \mathit{LetccTyp}^1 \qquad \frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma \vdash e_2 : \tau'\ \textbf{cont}}{\Gamma \vdash \textbf{throw}[\tau]\ e_1\ \textbf{to}\ e_2 : \tau}\ \mathit{ThrowTyp}$$

$$\frac{\cdot \vdash k : \tau\ \textbf{stack}}{\cdot \vdash \textbf{cont}(k) : \tau\ \textbf{cont}}\ \mathit{ContTyp}$$

[1] $x \notin \operatorname{dom}\Gamma$

Now since stacks k are represented by SML functions, there is no way, really, to typecheck an expression of the form $\textbf{cont}(k)$. Thus, you do not have to implement the rule *ContTyp*. You will need to implement the other two typing rules.

Now for the dynamic semantics. There is a new value, namely, $\textbf{cont}(k)$, where k is a stack. The C machine rules are

$$\begin{aligned}
&k > \textbf{letcc}[\tau]\ x\ \textbf{in}\ e \mapsto k > \{\textbf{cont}(k)/x\}e\\
&k > \textbf{throw}[\tau]\ e_1\ \textbf{to}\ e_2 \mapsto k \triangleright \textbf{throw}[\tau]\ \Box\ \textbf{to}\ e_2 > e_1\\
&k \triangleright \textbf{throw}[\tau]\ \Box\ \textbf{to}\ e_2 < v_1 \mapsto k \triangleright \textbf{throw}[\tau]\ v_1\ \textbf{to}\ \Box > e_2\\
&k \triangleright \textbf{throw}[\tau]\ v_1\ \textbf{to}\ \Box < \textbf{cont}(k') \mapsto k' < v_1
\end{aligned}$$

Q 3.1. [30 pts] As before, extend `typing.sml` and `eval.sml` to treat the new constructs. You do not have to implement the rule *ContTyp*, however.

§ 4. Test cases. You are responsible for creating some test cases to help verify that your implementation is working. Your test cases should include comments describing what is being tested and what the expected results are.

Q 4.1. [5 pts] Create four test cases that exercise these new constructs. Exactly one, `fail.mml`, should fail to typecheck, for an "interesting" reason. All of the others must be well typed. None of the test cases can loop forever. Name your test cases `1.mml`, `2.mml`, `3.mml`, and `fail.mml`. The test cases should be handed in together with your other files.