Assignment 3: Adding streams to MinML

15-312: Foundations of Programming Languages
Kevin Watkins ⟨kw@cmu.edu⟩

Out: Thursday, February 10, 2005
Due: Thursday, February 17, 2005 (10:30 a.m.)

50 points total

In this written assignment, you're asked to add recursive streams to MinML. You'll have to provide the static and dynamic semantics for the new constructs and prove type safety for the extended language.

MinML with recursive streams. Our new version of MinML is the language from Assignment 2 extended with the following constructs for recursive streams.

Construct	Concrete	Abstract (h.o.a.s.)
Empty stream	**nil**	**nil**()
Recursive cons	**cons rec** $x = e_1 \mathbin{\#} e_2$	**cons**$(x.\,e_1, x.\,e_2)$
Case of stream	**case** e **of nil** $\Rightarrow e_1 \mid x \mathbin{\#} y \Rightarrow e_2$	**case**$(e, e_1, x.\,y.\,e_2)$

Right now, you have no idea what these constructs mean. Below, we try to give some intuition for what they should mean by example. But part of your job for this assignment is to say, yourself, precisely what they mean, by giving static and dynamic semantics for them.

The recursive cons **cons rec** $x = e_1 \mathbin{\#} e_2$ constructs a stream with head e_1 and tail e_2. The quirk that makes a recursive stream different from an ML list, say, is that the variable x stands for the stream itself within e_1 and e_2 (so x is a bound variable with e_1 and e_2 as its scope). In this way a recursive stream can refer to itself within its definition. You can think of the symbol # as being analogous to ML's list constructor ::. If x doesn't actually occur in e_1 or e_2, then the recursive cons is more like a list cons, and we can abbreviate **cons rec** $x = e_1 \mathbin{\#} e_2$ as **cons rec** $\_ = e_1 \mathbin{\#} e_2$ or even just $e_1 \mathbin{\#} e_2$.

There is a new case construct for taking a stream apart. The meaning of

$$\textbf{case}\ e\ \textbf{of nil} \Rightarrow e_1 \mid x \mathbin{\#} y \Rightarrow e_2$$

is e_1 if e is **nil**, or e_2 if e is a cons. When it's a cons, the variables x and y stand for the head and tail of the stream. Their scope is e_2.

The operational semantics of streams is lazy; in **cons rec** $x = e_1 \mathbin{\#} e_2$, e_1 and e_2 are not evaluated until the value is needed. It's hard to see how recursive streams could be defined in a non-lazy way without leading to infinite looping in the examples below. To keep things simple, we only consider streams of integers; these have type **stream**.

Now we present a few examples intended to guide you in defining the *precise* static and dynamic semantics of recursive streams. In the informal descriptions we write $[x, y, z, \ldots]$ for the finite or infinite sequence of integers making up a stream.

$$\textbf{cons rec}\ x = 1 \mathbin{\#} (\textbf{cons rec}\ y = 2 \mathbin{\#} \textbf{nil})$$

The finite stream $[1, 2]$.

$$1 \mathbin{\#} 2 \mathbin{\#} \textbf{nil}$$

The same, abbreviated.

$$\mathit{ones} = (\textbf{cons rec}\ x = 1 \mathbin{\#} x)$$

The infinite stream $[1, 1, \ldots]$.

```
$\mathit{plus} = \textbf{fun}\ \mathit{plus}(s_1 : \textbf{stream}) : \textbf{stream} \rightarrow \textbf{stream}\ \textbf{is}$
  $\textbf{fun}\ p(s_2 : \textbf{stream}) : \textbf{stream}\ \textbf{is}$
    $\textbf{case}\ s_1\ \textbf{of}$
      $\textbf{nil} \Rightarrow \textbf{nil}$
      $\mid x_1 \mathbin{\#} s_1' \Rightarrow \textbf{case}\ s_2\ \textbf{of}$
        $\textbf{nil} \Rightarrow \textbf{nil}$
        $\mid x_2 \mathbin{\#} s_2' \Rightarrow (x_1 + x_2) \mathbin{\#} \mathit{plus}\ s_1'\ s_2'$
  $\textbf{end}$
$\textbf{end}$
```

Add finite or infinite streams $[x_1, y_1, \ldots]$ and $[x_2, y_2, \ldots]$, yielding $[x_1 + x_2, y_1 + y_2, \ldots]$.

```
$\mathit{map} = \textbf{fun}\ \mathit{map}(f : \textbf{int} \rightarrow \textbf{int}) : \textbf{stream} \rightarrow \textbf{stream}\ \textbf{is}$
  $\textbf{fun}\ m(s : \textbf{stream}) : \textbf{stream}\ \textbf{is}$
    $\textbf{case}\ s\ \textbf{of}$
      $\textbf{nil} \Rightarrow \textbf{nil}$
      $\mid x \mathbin{\#} s' \Rightarrow (f\ x) \mathbin{\#} (m\ s')$
  $\textbf{end}$
$\textbf{end}$
```

Map f over the finite or infinite stream $[x_1, x_2, \ldots]$, yielding $[f\,x_1, f\,x_2, \ldots]$.

$$\mathit{nats} = (\textbf{cons rec}\ s = 0 \mathbin{\#} (\mathit{plus}\ \mathit{ones}\ s))$$

The infinite stream $[0, 1, 2, 3, \ldots]$.

$$\mathit{fibs} = (\textbf{cons rec}\ s = 0 \mathbin{\#} (\textbf{cons rec}\ t = 1 \mathbin{\#} (\mathit{plus}\ s\ t)))$$

The Fibonacci numbers $[0, 1, 1, 2, 3, 5, 8, \ldots]$.

§ 1. Static semantics. The static semantics of the original MinML is shown in Figure 1 for your reference.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textit{VarTyp} \qquad \frac{}{\Gamma \vdash k : \mathbf{int}}\ \textit{NumTyp}$$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}\ \textit{TrueTyp} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}\ \textit{FalseTyp}$$

$$\frac{\Gamma \vdash e : \mathbf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{end} : \tau}\ \textit{IfTyp} \qquad \frac{\Gamma \vdash e_1 : \tau_{o1} \quad \dots \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \dots, e_n) : \tau_o}\ \textit{OpTyp}$$

$$\frac{\Gamma, (f : \tau_1 \to \tau_2), (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \mathbf{fun}\ f(x : \tau_1) : \tau_2\ \mathbf{is}\ e\ \mathbf{end} : \tau_1 \to \tau_2}\ \textit{FunTyp}^1 \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau}\ \textit{AppTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textit{LetTyp}^2 \qquad {}^1\ f, x \notin \operatorname{dom}\Gamma \qquad {}^2\ x \notin \operatorname{dom}\Gamma$$

Figure 1: Static semantics of the original MinML

Q 1.1. [5 pts] Write the full grammar of MinML types τ and expressions e extended with the new constructs. (Include only the "official" syntax, not the abbreviations.)

Q 1.2. [5 pts] Define the typing judgment $\Gamma \vdash e : \tau$ for each of the new syntactic constructs.

§ 2. Dynamic semantics. The dynamic semantics of the original MinML is shown in Figure 2 for your reference.

Q 2.1. [5 pts] Write the grammar of MinML values v extended with the new constructs.

Q 2.2. [5 pts] Complete the dynamic semantics for the new constructs. You must define a small-step semantics $e \mapsto e'$. The evaluation of **cons rec** must be lazy, and the semantics must be deterministic (but you don't have to prove this).

Q 2.3. [extra credit] Extend the definition of the C-machine $s \mapsto s'$ with the new constructs for streams.

§ 3. Type safety. You are free to make use of the following lemmas. You do not have to prove them. (But if these lemmas don't hold for your static semantics from Section 1, you must go back and change it so that they do.)

> **Weakening.** If $\Gamma \vdash e : \tau$ and $x \notin \operatorname{dom}\Gamma$, then $\Gamma, (x : \tau') \vdash e : \tau$.
>
> **Value substitution.** If $\cdot \vdash v : \tau'$ and $(x : \tau'), \Gamma \vdash e : \tau$ then $\Gamma \vdash \{v/x\}e : \tau$.

If you use any lemmas other than the ones above, you must state and prove them too.

Q 3.1. [30 pts] State and prove the preservation and progress theorems for your new language. You can skip any cases for rules and constructs that haven't changed from the original MinML (but your proofs must be structured in such a way that the skipped cases work exactly the way they were done in lecture).

$$\frac{e_i \mapsto e_i'}{o(v_1, \ldots, e_i, \ldots, e_n) \mapsto o(v_1, \ldots, e_i', \ldots, e_n)}\ \mathit{OpArg}$$

$$\frac{\text{(by primop } o\text{)}}{o(v_1, \ldots, v_n) \mapsto v}\ \mathit{OpVals}$$

$$\frac{e \mapsto e'}{\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end} \mapsto \textbf{if } e' \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ end}}\ \mathit{IfCond}$$

$$\frac{}{\textbf{if true then } e_1 \textbf{ else } e_2 \textbf{ end} \mapsto e_1}\ \mathit{IfTrue}$$

$$\frac{}{\textbf{if false then } e_1 \textbf{ else } e_2 \textbf{ end} \mapsto e_2}\ \mathit{IfFalse}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathit{AppFun} \qquad \frac{e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}\ \mathit{AppArg}$$

$$\frac{(v_1 = \textbf{fun } f(x : \tau_1) : \tau_2 \textbf{ is } e \textbf{ end})}{v_1\ v_2 \mapsto \{v_1, v_2 / f, x\}e}\ \mathit{AppCall}$$

$$\frac{e_1 \mapsto e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \mapsto \textbf{let } x = e_1' \textbf{ in } e_2 \textbf{ end}}\ \mathit{LetArg}$$

$$\frac{}{\textbf{let } x = v_1 \textbf{ in } e_2 \textbf{ end} \mapsto \{v_1/x\}e_2}\ \mathit{LetVal}$$

Figure 2: Dynamic semantics of the original MinML

Be sure to show all necessary steps in your proofs—the model solution for Assignment 1, available from the course web page, has more information on what "necessary" means.