CPSC 429/529: Functional Programming
Carsten Schürmann
Date: Wed 31 Jan 2001

Homework 2

due Wed 14 Feb 2001 at 9:30am

In this assignment, you will implement a simple interpreter for a tiny lazy functional language. The language has primitive lists and integers, and not much else. The following table gives a summary of the language syntax.

Feature	Values V	Programs P
functions	$\lambda x. P$	$P_1\ P_2 \quad x \quad V$
recursion		$\mu x. P$
integers	$\overline{n}$	**add** $P_1\ P_2$ **ifzero** P_0 **then** P_1 **else** P_2
lists	$[\,] \quad P_1 : P_2$	**head** P **tail** P
		case P_0 **of** $[\,] \to P_1$ **or** $h : t \to P_2$

As an example, consider fetching the nth element from a list. In Haskell, we might write this function as follows:

```
nth :: Int -> [a] -> a
nth 0 (x:_)  = x
nth n (_:xs) = nth (n-1) xs
```

Our little language does not support pattern matching, so we must use **ifzero, head,** and **tail.** Furthermore, recursive functions must be expressed using the fixpoint operator μ. Here is an equivalent program:

$$\text{nth} = \mu g.\, \lambda n.\, \lambda l.\, \textbf{ifzero}\ n\ \textbf{then}\ \textbf{head}\ l\ \textbf{else}\ g\ (\textbf{add}\ n\ \text{-}1)\ (\textbf{tail}\ l)$$

The complete operational semantics for this language are given in Figure 1. Evaluation is formalized using the judgment $E \vdash P \hookrightarrow C$, where E is an environment, P is a program, and C is a *closure.* The environment E contains bindings of the form $x \mapsto C$, mapping each free variable x in P to a closure C. The empty environment is written $\circ$. It is used, for example, in closures with integer values (see rule `ev-add`). A closure is simply a pair $\{E; P\}$ of an environment and a program. The environment provides bindings for the free variables in the program. If the contents of the environment are irrelevant, we write $\{\_; P\}$.

There are four parts to this assignment.

1. (20 points) Design Haskell data types for representing values and programs. Call them `Val` and `Prog`. Variables can be represented as strings.

$$\frac{}{E \vdash V \hookrightarrow \{E; V\}} \quad \textsf{(ev-val)}$$

$$\frac{E(x) = \{E'; P\} \qquad E' \vdash P \hookrightarrow C}{E \vdash x \hookrightarrow C} \quad \textsf{(ev-var)}$$

$$\frac{\begin{array}{l} E \vdash P_1 \hookrightarrow \{E'; \lambda x.\, P\} \\ E', x \mapsto \{E; P_2\} \vdash P \hookrightarrow C \end{array}}{E \vdash P_1\ P_2 \hookrightarrow C} \quad \textsf{(ev-app)}$$

$$\frac{\begin{array}{l} E \vdash P_1 \hookrightarrow \{\_; \overline{n_1}\} \\ E \vdash P_2 \hookrightarrow \{\_; \overline{n_2}\} \end{array}}{E \vdash \mathbf{add}\ P_1\ P_2 \hookrightarrow \{\circ; \overline{n_1 + n_2}\}} \quad \textsf{(ev-add)}$$

$$\frac{\begin{array}{l} E \vdash P_0 \hookrightarrow \{\_; \overline{0}\} \\ E \vdash P_1 \hookrightarrow C \end{array}}{E \vdash \mathbf{ifzero}\ P_0\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \hookrightarrow C} \quad \textsf{(ev-ifz)}$$

$$\frac{\begin{array}{l} E \vdash P_0 \hookrightarrow \{\_; \overline{n}\} \qquad n \neq 0 \\ E \vdash P_2 \hookrightarrow C \end{array}}{E \vdash \mathbf{ifzero}\ P_0\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \hookrightarrow C} \quad \textsf{(ev-ifnz)}$$

$$\frac{\begin{array}{l} E \vdash P_0 \hookrightarrow \{E'; P_1 : P_2\} \\ E' \vdash P_1 \hookrightarrow C \end{array}}{E \vdash \mathbf{head}\ P_0 \hookrightarrow C} \quad \textsf{(ev-hd)}$$

$$\frac{\begin{array}{l} E \vdash P_0 \hookrightarrow \{E'; P_1 : P_2\} \\ E' \vdash P_2 \hookrightarrow C \end{array}}{E \vdash \mathbf{tail}\ P_0 \hookrightarrow C} \quad \textsf{(ev-tl)}$$

$$\frac{\begin{array}{l} E \vdash P_0 \hookrightarrow \{E'; P_H : P_T\} \\ E, h \mapsto \{E'; P_H\}, t \mapsto \{E'; P_T\} \\ \quad \vdash P_C \hookrightarrow C \end{array}}{\begin{array}{l} E \vdash \mathrm{case}\ P_0\ \mathrm{of}\ [\,] \to P_N \\ \qquad\qquad \mathrm{or}\ h : t \to P_C\ \hookrightarrow C \end{array}} \quad \textsf{(ev-casecons)}$$

$$\frac{E \vdash P_0 \hookrightarrow \{\_; [\,]\} \qquad E \vdash P_N \hookrightarrow C}{\begin{array}{l} E \vdash \mathrm{case}\ P_0\ \mathrm{of}\ [\,] \to P_N \\ \qquad\qquad \mathrm{or}\ h : t \to P_C\ \hookrightarrow C \end{array}} \quad \textsf{(ev-casenil)}$$

$$\frac{E, x \mapsto \{E; \mu x.\, P\} \vdash P \hookrightarrow C}{E \vdash \mu x.\, P \hookrightarrow C} \quad \textsf{(ev-fix)}$$

Figure 1: Operational semantics.

2. (10 points) Decide on a representation `Env` for environments. Implement a function `get` to find a variable in an environment, and `put` to add a binding to an environment. Perhaps they will have the following types:

```
get :: String -> Env -> (Env, Prog)
put :: String -> (Env, Prog) -> Env -> Env
```

   where `(Env, Prog)` is the representation of a closure. Your `Env` representation need not be fancy or fast; a linear search through a list of bindings is acceptable. You will use `get` to implement (ev-var); `put` is used in (ev-app), (ev-casecons), and (ev-fix).

3. (50 points) Implement the evaluation judgment $E \vdash P \hookrightarrow C$ as a Haskell function `eval`. It will take E and P as arguments and return the closure C.

   You may have noticed an important invariant about the evaluation judgment. The closure that is returned always has the form $\{E; V\}$; that is, the closure always contains a value. One way to enforce this invariant in your code is to give `eval` the following type:

```
eval :: Env -> Prog -> (Env, Val)
```

   Suppose that a program `p`, when run in an initial environment `e`, returns a list of integers. We would evaluate the program like this:

```
(e', v) = eval e p
```

   The value `v` is then either $[\,]$ or a cons cell $(P_H : P_T)$ where the head and tail might require further evaluation. In order to see all the integers in the list, we need to recursively evaluate all the sub-components.

   Implement a top-level function `toIntList` to convert the result to a Haskell integer list, evaluating more of the program as necessary.

```
toIntList :: (Env, Val) -> [Int]
```

   Now, you can run a program `p` and see its result as a list of integers by calling `toIntList (eval e p)`. Note that if `p` is non-terminating, `toIntList` will loop forever!

4. (20 points) Translate the Haskell program for computing the Fibonacci sequence using an infinite stream (Hudak, page 191) to our little language and encode it using your data types. You will also need to implement 'zip' and 'take'. You can hard-code the addition in your 'zip' function; it need not take the function `(+)` as an argument. See Lecture 10 on the web for implementations of these functions in Haskell.

   Finally, create an initial environment and a program that will take the first 15 numbers from the infinite Fibonacci sequence. Use your evaluator to run the program.