Lecture Notes on
Lazy Records and Mutable Store

15-814: Types and Programming Languages
Frank Pfenning

Lecture 22
Tuesday, November 17, 2020

1 Introduction

We have moved from a semantics directly on expressions to one that makes memory explicit and supports concurrency (at the discretion of the scheduler or the language designer). Memory is allocated and then written to at most once; after that it may be read many times.

In imperative languages we can also mutate the contents of a memory cell by writing a different value to it. In a functional language, this is typically segregated, either in a monad (as in Haskell) or via a new type of mutable references (as in ML). We pursue here the latter approach because there is a slightly lower conceptual overhead.

Before that, we consider some examples of stream programming, which is a good example of lazy functional programming which is available to use most easily via lazy pairs and (more generally) lazy records.

2 Lazy Records and Streams

A *lazy record* is a generalization of a lazy pair where each alternative has a different label i. They were introduced in Exercise L20.1 with the syntax

$$\begin{array}{lll} \text{Types} & ::= & \ldots \mid \&_{i \in I}(i : \tau_i) \\ \text{Expressions} & ::= & \ldots \mid \langle\!| i \Rightarrow e_i |\!\rangle_{i \in I} \mid e \cdot j \end{array}$$

where $\langle\!| i \Rightarrow e_i |\!\rangle_{i \in I}$ has type $\&_{i \in I}(i : \tau_i)$ if each e_i has type τ_i. Similarly, for an e of type $\&_{i \in I}(i : \tau_i)$ the projection onto j (written $e \cdot j$) has type τ_j.

As an example, consider potentially infinite streams *stream* α of elements of some type α may be defined as

$$\textit{stream}\ \alpha = \rho s.\, (\mathbf{hd} : \alpha)\ \&\ (\mathbf{tl} : s)$$

which then satisfies

$$\textit{stream}\ \alpha \cong (\mathbf{hd} : \alpha)\ \&\ (\mathbf{tl} : \textit{stream}\ \alpha)$$

The concrete LAMBDA syntax is quite similar. Since we don't have type constructors in the implementation, we just consider streams of natural numbers which are defined with

```
type stream = $stream. ('hd : nat) & ('tl : stream)
```

Next we would like to produce an infinite stream of increasing numbers. The specification is that

$$\textit{up}\ n = n, n+1, n+2, \ldots$$

which we write as

```
decl up : nat -> stream
defn up = $up. \n. fold (| 'hd => n | 'tl => up (succ n) |)
```

Here we see the concrete syntax `(| 'i1 => e1 | ... | 'in => en |)` for a lazy record with fields `'i1` through `'in`. Laziness of the records is crucial here because otherwise the function $\textit{up}\,\overline{n}$ would never terminate. Indeed, evaluating

```
eval s0 = up zero
```

just produces (after 2 evaluation steps) a stream we cannot observe:

```
% eval s0 = up zero
% 2 evaluation steps
decl s0 : stream
defn s0 = fold ---
```

Fortunately, we can write a function to observe the first n elements of a stream as a list. For this purpose we define lists, restricting ourselves to the special case of lists of natural numbers.

```
type list = $list. ('nil : 1) + ('cons : nat * list)
decl nil : list
decl cons : nat -> list -> list
defn nil = fold 'nil ()
defn cons = \x. \l. fold 'cons (x, l)
```

Our specification is now that

$$\textit{take}\ n\ s = [s_1, \ldots, s_n]$$

where s_i is the i^{th} element of the stream. We start by cases over n, and returning the empty list if n is zero.

```
1 decl take : nat -> stream -> list
2 defn take = $take. \n. \s.
3    case n
4      of ( fold 'zero () => nil
5         | fold 'succ m => ... )
```

In the case where $n = m + 1$ we would like to create a list of length with the first element begin the head of the stream (obtained with `(unfold s).'hd`). The remainder of the list is the result of a recursive call to take m elements from the tail of the stream.

```
1 decl take : nat -> stream -> list
2 defn take = $take. \n. \s.
3    case n
4      of ( fold 'zero () => nil
5       | fold 'succ m => cons ((unfold s).'hd)
6                              (take m ((unfold s).'tl)) )
```

Then, taking the first 5 elements from the stream $0, 1, 2, \ldots$ eis achieved with

```
1 eval l5 = take _5 (up _0)
```

which indeed yields the list $[\overline{0}, \ldots, \overline{4}]$.

As the next programming puzzle we would like to compute the stream of Fibonacci numbers, $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The key insight is that we need to remember *two* numbers to generate the next one, generalizing the idea behind *up*. We specify

$$\textit{fib}\ n\ k = n, k, n + k, k + (n + k), \ldots$$

after which the actual Fibonacci sequence is *fib* $0\ 1$. We implement this function by shifting the second argument k to become the first argument in the recursive call, and the sum $n + k$ to become the new second argument.

```
1 decl fib : nat -> nat -> stream
2 defn fib = $fib. \n. \k.
3    fold (| 'hd => n | 'tl => fib k (plus n k) |)
4
5 eval fib_stream = fib _0 _1
```

Records here are lazy so, as before, `fib_stream` is not observable. But we can test our function by taking the first 10 elements with

```
eval f10 = take (plus _5 _5) fib_stream
```

There are more examples of stream programming in Exercise 2.

3 Object-Oriented Programming

We can also use lazy records to model some idioms from object-oriented programming. Consider an object of type *stack* which can receive two messages: pushing another element onto the stack, or popping an element from the stack. In the latter case, the response is either **none** (the stack is empty) or **some** and the element.

```
type stack = $stack. ('push : nat -> stack)
                   & ('pop :   ('none : stack)
                             + ('some : nat * stack))
```

The implementation of the stack maintains a list in the local state: it adds a new element to the front of the list to implement **push** and deconstructs the list to implement **pop**. Note that the methods of the objects are the components of a lazy record.

```
decl stack_list : list -> stack
defn stack_list = $stack_list. \l. fold
    (| 'push => \x. stack_list (cons x l)
     | 'pop => case l
                 of ( fold 'nil () => 'none (stack_list l)
                    | fold 'cons (x,l') => 'some (x, stack_list l') )
     |)

decl stack_new : stack
defn stack_new = stack_list nil
```

See the file `lazy.cbv` for a illustrative sequence of push and pop operations.

4 Streams and Functions

An excellent question was raised in lecture, namely if any stream can be represented by a function of type $nat \to nat$ and vice versa. Assuming totality of the functions involved, we were able to conjecture

$$nat \to nat \cong stream$$

using the following functions

```
1  decl forth : (nat -> nat) -> stream
2  decl back : stream -> (nat -> nat)
3
4  % forth f = f 0, f 1, f 2, ...
5  % forth' f n = f n, f (n+1), f (n+1), ...
6  decl forth' : (nat -> nat) -> nat -> stream
7  defn forth' = $forth'. \f. \n.
8     fold (| 'hd => f n | 'tl => forth' f (succ n) |)
9  defn forth = \f. forth' f zero
10
11 defn back = $back. \s. \n.
12   case n
13     of ( fold 'zero () => (unfold s).'hd
14        | fold 'succ m => back ((unfold s).'tl) m )
```

Some small examples in `lazy.cbv` seemed to confirm the correctness of these definitions.

5 The Type of Mutable References

Returning to our original goal of this lecture, we now consider mutable references. We will have to depart from the strong logical basis of our language, but the notation and concepts we have developed to describe typing are sufficient to easily capture the statics and dynamics of the new constructs.

We introduce one new type constructor and three new forms of expression into our functional language:

$$\begin{array}{llcl} \text{Types} & \tau & ::= & \ldots \mid \mathsf{ref}\ \tau \\ \text{Expressions} & e & ::= & \ldots \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid\ !e \end{array}$$

Operationally, ref e evaluates e to a value v, then creates a new mutable reference m and initializes its value to v. An assignment $e_1 := e_2$ evaluates e_1 to a mutable reference m, then e_2 to a value v_2 and stores v_2 in m. It returns just the unit element, since its principal task is the effect on m. Finally, $!e$ (which has nothing to do with ! to denote persistent semantic objects) evaluates e to a reference m and returns the current value of m. Based on

this description, we type these new expressions as follows

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau}\ \mathsf{tp/ref} \qquad \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}\ \mathsf{tp/assign}$$

$$\frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash\ !e : \tau}\ \mathsf{tp/deref}$$

These rules do not fit the previous patterns of constructor and destructors because of the rule for mutation tp/assign.

It seems difficult, if not impossible, to specify the semantics of mutable references directly on expressions in the style we have done before. Fortunately, we already have a semantics with an explicit store so we can update that. The textbook instead generalizes the small-step semantics for expression by adding a single store μ and now stepping $\mu \parallel e \mapsto \mu' \parallel e'$ [Har16, Chapters 34 & 35].

6 Translation to Our Concurrent Language

We exploit the fact we already have a representation of memory in this translation, and only two small twists are necessary. Warning: some of what is below we will later find out is not quite right. We write m for the address of a mutable cell.

$$\begin{array}{l}[\![\mathsf{ref}\ e]\!]\, d = m \leftarrow [\![e]\!]\, m\ ; \\ \qquad\qquad d^W.\mathsf{addr}(m)\end{array}$$

Here, we introduce a new form of value, $\mathsf{addr}(m)$ which denotes the address of a mutable cell, here m. This value is deposited in destination d as required.

Reading from a mutable destination is simple.

$$\begin{array}{l}[\![!e]\!]\, d = x \leftarrow [\![e]\!]\, x\ ; \\ \qquad \mathsf{case}\ x^R\ (\mathsf{addr}(m) \Rightarrow d^W \leftarrow m^R)\end{array}$$

Finally, mutating a cell. At first we might try

$$\begin{array}{lll}[\![e_1 := e_2]\!]\, d = x_1 \leftarrow [\![e_1]\!]\, x_1\ ; & & \\ \qquad\qquad x_2 \leftarrow [\![e_2]\!]\, x_2\ ; & & \\ \qquad\qquad \mathsf{case}\ x_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow x_2^R) & \% \textit{ bug here!} & \end{array}$$

The problem here is that the translation of $e_1 := e_2$ is supposed to write to destination d, but does not do so. Recall that we decreed that the assignment should return the unit element, so we might write

$$\begin{aligned}[\![e_1 := e_2]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, x_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, x_2 \ ; \\ & \mathsf{case}\ x_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \leftarrow x_2^R \ ; \\ & \qquad\qquad\qquad\qquad d.\langle\,\rangle)\end{aligned}$$

However, this requires a version of the copy process that allows a continuation. Let's write this as $m^W \Leftarrow d_2^R$, and we get

$$\begin{aligned}[\![e_1 := e_2]\!]\, d = {} & d_1 \leftarrow [\![e_1]\!]\, d_1 \ ; \\ & d_2 \leftarrow [\![e_2]\!]\, d_2 \ ; \\ & \mathsf{case}\ d_1^R\ (\mathsf{addr}(m) \Rightarrow m^W \Leftarrow d_2^R \ ; \\ & \qquad\qquad\qquad\qquad d.\langle\,\rangle)\end{aligned}$$

The new process expression has the dynamics

$$!\mathsf{cell}\ m\ W, !\mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R\ ;\ P) \mapsto !\mathsf{cell}\, m\ W', \mathsf{proc}\ d\ P \quad \%\ \textit{bug!}$$

Writing this out, however, we notice a second problem: the cell m has to be ephemeral. If it were persistent, then after this transition m would have two values: W and W'.

We can fix this in two ways. Either we make all cells (mutable or not) ephemeral. This means we have to revisit all the rules so far and make sure cell are not consumed when they are read but carried over. Alternatively, we can make only mutable cells ephemeral and keep all others persistent. Let's use the first approach. We modify the rules at the end of Section L21.4 by dropping the ! everywhere. Where we match against $!\mathsf{cell}\ c\ W$ on the left-hand side, we just replace it by $\mathsf{cell}\ c\ W$ and repeat it on the right-hand side. The rule for the new "write" construct becomes

$$\mathsf{cell}\ m\ W, \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ (m^W \Leftarrow c^R\ ;\ P) \mapsto \mathsf{cell}\ m\ W', \mathsf{cell}\ c\ W', \mathsf{proc}\ d\ P$$

For the other approach, see Exercise 1.

7 Race Conditions

In the presence of mutable references, sequential computation proceeds as before, scheduling such that in $x \leftarrow P\ ;\ Q$ the process P completes (and therefore writes to x) before Q starts. This also means that the read and write operations on mutable cells have a well-defined order.

Under the concurrent semantics, however, the picture is more complicated. Consider the following expression:

$$(\lambda x.\, \langle x := \mathit{succ}\ x, \langle x := \mathit{succ}\ x, !x\rangle\rangle)\ (\mathsf{ref}\ \mathit{zero})$$

The value of this expression will be

$\langle\langle\rangle, \langle\langle\rangle, n\rangle\rangle$

where $n \in \{\overline{0}, \overline{1}, \overline{2}\}$. For example, we obtain $\overline{0}$ if $!x$ executes before the increments. Note also that either increment or dereference of the value might have to wait until the initialization of the mutable cell with $\overline{0}$ completes because the body of the function can execute in parallel with the argument.

Despite these difficulties, progress and preservation theorems continue to hold, but it becomes much more difficult to reason about the correctness of programs. Similarly, we don't lose all of parametricity, but logical equality (and, more generally, logical relations) now require *step-indexing* [AM01, TTA+13].

Exercises

Exercise 1 Provide an alternative dynamics for our language with mutable cells, where regular cells become persistent once written, while mutable cells are ephemeral. You may have to introduce some new kinds of semantic objects or some new forms of process expression, or both.

Exercise 2 Write functions on streams as in Section 2 satisfying the specifications below.

(i) *alt* : $\forall\alpha.\, \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha$ which alternates the elements from the two streams, starting with the first element of the first stream.

(ii) *filter* : $\forall\alpha.\, (\alpha \to \mathit{bool}) \to \mathit{stream}\ \alpha \to \mathit{stream}\ \alpha$ which returns the stream with just those elements of the input stream that satisfy the given predicate.

(iii) *map* : $\forall\alpha.\, \forall\beta.\, (\alpha \to \beta) \to (\mathit{stream}\ \alpha \to \mathit{stream}\ \beta)$ which returns a stream with the result of applying the given function to every element of the input stream.

(iv) *diag* : $\forall\alpha.\, \mathit{stream}\ (\mathit{stream}\ \alpha) \to \mathit{stream}\ \alpha$ which returns a stream consisting of the first element of the first stream, the second element of the second stream, the third element of the third stream, etc.

You may use earlier functions in the definition of later ones and write auxiliary functions as needed.

In the LAMBDA implementation, you may choose the special case that $\alpha = \beta = \mathit{nat}$. In the absence of type constructors in LAMBDA, define types

$\mathit{stream} = \rho s.\, (\mathbf{hd} : \mathit{nat}) \,\&\, (\mathbf{tl} : s)$
$\mathit{sstream} = \rho ss.\, (\mathbf{hd} : \mathit{stream}) \,\&\, (\mathbf{tl} : ss)$

where the first was already present in Section 2) and the second is needed for part (iv).

Your functions should be such that only as much of the output stream is computed as necessary to obtain a *value* of type *stream* α but not the components contained in the lazy record. For example, among the three definitions below of a stream transducer that adds 1 to every element, only the first definition would be lazy enough. The second definition (succs') would be still terminating, but slighty too eager (for example, we may never access the element at the head of the resulting stream which would have been computed unnecessarily), while the third (succs'') would not even be terminating any more.

$\mathit{succs} : \mathit{stream}\ \mathit{nat} \to \mathit{stream}\ \mathit{nat}$
$\mathit{succs} = \lambda s.\, \langle\!|\mathtt{hd} \Rightarrow \mathit{succ}\,((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow \mathit{succs}\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl})|\!\rangle$

$\mathit{succs}' = \lambda s.\ \mathsf{let}\ x = \mathit{succ}\ ((\mathsf{unfold}\ s) \cdot \mathtt{hd})$
$\qquad\qquad \mathsf{in}\ \langle\!|\mathtt{hd} \Rightarrow x, \mathtt{tl} \Rightarrow \mathit{succs}'\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl})|\!\rangle$

$\mathit{succs}'' = \lambda s.\ \mathsf{let}\ s' = \mathit{succs}''\ ((\mathsf{unfold}\ s) \cdot \mathtt{tl})$
$\qquad\qquad \mathsf{in}\ \langle\!|\mathtt{hd} \Rightarrow \mathit{succ}\ ((\mathsf{unfold}\ s) \cdot \mathtt{hd}), \mathtt{tl} \Rightarrow s'|\!\rangle$

Here we have used $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ as syntactic sugar for $(\lambda x.\, e)\, e'$.

Exercise 3 Following the style of object-oriented programming in Section 3 consider the types of queue

$$\begin{array}{lll} \mathit{queue}\ \alpha = \rho s. & & (\mathbf{enq} : \alpha \to \mathit{queue}\ \alpha) \\ & \& (\mathbf{deq} : & (\mathbf{none} : \mathit{queue}\ \alpha) \\ & & + (\mathbf{some} : \alpha \times \mathit{queue}\ \alpha)) \end{array}$$

(i) Write a function

$$\mathit{reverse} : \forall \alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha$$

that reverses the elements of the given stack.

(ii) Provide an implementation of queues

$$\mathit{queue\_stacks} : \forall \alpha.\, \mathit{stack}\ \alpha \to \mathit{stack}\ \alpha \to \mathit{queue}\ \alpha$$

where a queue is represented by a pair of stacks (see below).

(iii) Provide an empty queue

$$\mathit{queue\_new} : \forall\alpha.\, \mathit{queue}\ \alpha$$

One of your stacks should be the *input stack*. Elements to be enqueued should be pushed on this input stack. The second stack should be the *output stack*. Elements to be dequeued should be taken from the output stack. If the output stack happens to be empty but some elements remain on the input stack, reverse the input stack to become the new output stack. This technique is sometimes called *functional queues*.

As in Section 3, in the absence of type constructors in LAMBDA you may specialize the types of stacks and queues to $\alpha = \mathit{nat}$.

Exercise 4 Streams, as we have defined them in this lecture, do not memoize their results, which means that if we repeatedly access the tail or head of a stream it may be recalculated each time. Using mutable references, define memoizing streams that avoids this recomputation and illustrate it through some examples.

References

[AM01] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *Transactions on Programming Languages and Systems*, 23(5):657–683, 2001.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[TTA+13] Aaron Joseph Turon, Jacob Thamsbord, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *Symposium on Principles of Programming Languages (POPL'13)*, pages 343–356, Rome, Italy, January 2013. ACM.