Lecture Notes on Futures

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 23
November 13, 2003

In this lecture we first examine a technique to specify the operational semantics for lazy evaluation. This is an implementation technique for a call-by-name semantics that avoids re-evaluating expressions multiple times by memoizing the result of the first evaluation. Then we use a similar technique to specify the meaning of *futures*, a construct that introduces parallelism into evaluation. Futures were first developed for Multilisp, a dynamically typed, yet statically scoped version of Lisp specifically designed for parallel computation. A standard reference on futures is:

> Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems, 7(4):501-538, October 1985.

One advantage of call-by-name function application over call-by-value is that it avoids the work of evaluating the argument if it is never needed. More broadly, lazy constructors avoid work until the data are actually used. In turn, this has several drawbacks. One of them is that the efficiency model of such a language is more difficult to understand than for a call-by-value language. The second is that lazy constructors introduce infinite values of data types which complicate inductive reasoning about programs. However, the most obvious problem is that if an expression is used several times it will be computed several times unless we can find an implementation technique to avoid this.

There are two basic approaches to avoid re-evaluation of the argument of a function application. The first is to analyze the function body to determine if the argument is really needed. If so, we evaluate it eagerly and then work with the resulting value. This is semantically transparent, but there are many cases where we cannot tell statically if an argument will be needed. The other is to create a so-called *thunk*[1] and pass a reference to the thunk as the actual argument. When the argument is needed we evaluate the thunk and memoize the resulting value. Further reference to the thunk now just returns the value instead of evaluating it again. Note that this strategy is only a correct implementation of call-by-name if there are no effects in the language (or, if there are effects, they are encapsulated in a monad).

We can think of a thunk as a reference that we can write only once (the first time it is accessed) and henceforth will continue to be the same value. So our semantic specification for lazy evaluation borrows from the ideas in the operational semantics of mutable references. We generalize the basic judgment $e \mapsto e'$ to $\langle H, e\rangle \mapsto \langle H', e'\rangle$ where H and H' contains all thunks, and e and e' can refer to them by their labels.

$$\textit{Thunks} \qquad H ::= \cdot \mid H, l{=}e$$

Note thunks may be expressions; after they have been evaluated the first time, however, they will be replaced by values. First, the rules for call-by-name application.

$$\frac{\langle H, e_1\rangle \mapsto \langle H', e_1'\rangle}{\langle H, \texttt{apply}(e_1, e_2)\rangle \mapsto \langle H', \texttt{apply}(e_1', e_2)\rangle}$$

$$\frac{}{\langle H, \texttt{apply}(\texttt{fn}(\tau, x.e_1), e_2)\rangle \mapsto \langle (H, l{=}e_2), \{l/x\}e_1\rangle}$$

In the second rule, the label l must be new with respect to H. When the value of l is actually accessed, we need to force the evaluation of the thunk and then record that value.

$$\frac{\langle (H_1, l{=}e, H_2), e\rangle \mapsto \langle (H_1', l{=}e^*, H_2'), e'\rangle}{\langle (H_1, l{=}e, H_2), l\rangle \mapsto \langle (H_1', l{=}e', H_2'), l\rangle}$$

$$\frac{v\ \ \mathsf{value}}{\langle (H_1, l{=}v, H_2), l\rangle \mapsto \langle (H_1, l{=}v, H_2), v\rangle}$$

Note that in the first rule, the result e^* must actually be equal to e. If it were not, that means the evaluation of e would actually require the thunk

[1]The name is a whimsical past tense of *think* derived from "something that has been thought of before".

l, which would lead to an infinite loop. This particular form of infinite loop is called a *black hole* can be detected, while other forms of non-termination remain.

It is left as an exercise to extend the statements of progress and preservation, or to show in which sense the call-by-name semantics coincides with the lazy evaluation semantics. Note also that there are other rules that can create thunks: essentially every time we need to substitute for a variable. We show one of these cases, namely recursion.

$$\overline{\langle H, \mathtt{rec}(\tau, x.e)\rangle \mapsto \langle (H, l{=}\{l/x\}e), l\rangle}$$

As an example of a black hole, consider $\mathsf{fix}\, f.f$. As an example of an expression that is *not* a black hole, yet fails to terminate consider $(\mathsf{fix}\, f.\lambda y.f\,(y+1))\,1$. It is instructive to simulate the execution of this expression.

$$\begin{array}{ll}
 & \langle \cdot, (\mathsf{fix}\, f.\lambda y.f\,(y+1))1\rangle \\
\mapsto & \langle (l = \lambda y.l\,(y+1)), l\,1\rangle \\
\mapsto & \langle (l = \lambda y.l\,(y+1)), (\lambda y.l(y+1))\ 1\rangle \\
\mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1), l\,(l_1+1)\rangle \\
\mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1), (\lambda y.l(y+1))\,(l_1+1)\rangle \\
\mapsto & \langle (l = \lambda y.l\,(y+1), l_1 = 1, l_2 = l_1+1), l\,(l_2+1)\rangle \\
\mapsto & \ldots
\end{array}$$

In order to detect black holes and take appropriate action we would allow thunks of the form $l{=}\bullet$ and replace the first rule by

$$\frac{\langle (H_1, l{=}\bullet, H_2), e\rangle \mapsto \langle (H_1', l{=}\bullet, H_2'), e'\rangle}{\langle (H_1, l{=}e, H_2), l\rangle \mapsto \langle (H_1', l{=}e', H_2'), l\rangle}$$

$$\overline{\langle (H_1, l{=}\bullet, H_2), l\rangle \mapsto \langle (H_1, l{=}\bullet, H_2), \mathsf{BlackHole}\rangle}$$

where BlackHole is a new error expression that must be propagated to the top level as shown in a previous lecture on run-time exceptions and errors.

Next we consider *futures*. The idea is that an expression $\mathsf{future}(e)$ spawns a parallel computation of e while returning immediately a pointer to the resulting value. If the resulting value is ever actually needed we say we are *touching* the future. When we touch the future we block until the parallel computation of its value has succeeded. However, in most situations we can pass around the future, construct bigger values, etc.

There are two principal differences to lazy evaluation as shown above. The first is that a future is treated as a value. This is important because unlike lazy evaluation, we are here in a call-by-value setting. Secondly, the computation of the future may proceed asynchronously, instead of being completed in full exactly the first time it is accessed. However, it is similar in the sense that once a future has been computed, its value is available everywhere it is referenced.

The typing rule for futures in source programs is exceedingly simple, since we consider it futures related only to how a program executes (sequentially or in parallel), but not what it computes.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{future}(e) : \tau}$$

Process labels l that arise during computation are given types just as stores or heaps are given types. Moreover, labels l are treated as values, which forces us to refine the value inversion lemma if we want to prove the progress theorem.

To describe such a computation we have to describe the overall state of all the computing threads. For this, we just use H, as defined above.

$$\text{Processes} \qquad H ::= \cdot \mid H, l{=}e$$

In this interpretation, labels l are thread identifiers, and $l{=}v$ represents a finished thread. So overall computation proceeds as in

$$H \mapsto H'$$

which non-deterministically selects a process that can proceed (that is, not finished or blocked) and makes a step. The judgment of making a step in the network of parallel processes is

$$\langle H, e\rangle \mapsto \langle H', e'\rangle$$

where H' may contain a new thread spawned by the step of e. Unlike lazy evaluation, this judgment cannot change any binding in H; this is reserved for the primary judgment. We start the overall computation of an expression e as a single process $l_0{=}e$ and we are finished when we have reached a state where *all* processes have the form $l{=}v$.

In order to be able to prove a progress theorem, we would like to maintain an order between the processes which reflects possible dependencies. That is, a process can refer to labels on its left, but not to itself or processes to its right.

The first rule non-deterministically selects a thread to perform a step. In this setting, a process can never refer to itself, because we have no recursive futures. Of course, we may have futures whose computation is recursive.

$$\frac{\langle H_1, e\rangle \mapsto \langle H_1', e'\rangle}{(H_1, l{=}e, H_2) \mapsto (H_1', l{=}e', H_2)} \qquad \frac{}{l \ \mathsf{value}}$$

The rules for the judgment $\langle H, e\rangle \mapsto \langle H', e'\rangle$ are the usual call-by-value rules, threading through H. It is only changed or referenced in the following two rules.

$$\frac{v\ val}{\langle (H_1, l{=}v, H_2), l\rangle \mapsto \langle (H_1, l{=}v, H_2), v\rangle} \qquad \frac{}{\langle H, \mathsf{future}(e)\rangle \mapsto \langle (H, l{=}e), l\rangle}$$

Because l is a value, it can be passed around, or looked up (in case the thread l has finished). This introduces some local non-determinism into expressions such as $\mathtt{apply}(l, e)$ because l could be looked up, or e could be reduced. In the end, the difference is not observable in a call-by-value language without effects. It could also be removed with some additional machinery, but we do not pursue this here, since non-determinism remains anyway due to the selection of the process to step.

Notice that an expression such as $\mathtt{apply}(l, v)$ is blocked until the thread computing l can completed. This is because it not a value, yet cannot be reduced.

The process selection rule must be prescient in this formulation, because we must traverse a thread expression to see if it is finished, can make a step, or is blocked, waiting for another thread to finish. This is a feature generally true for a small-step semantics with search rules. In a semantics with an evaluation stack, this can be avoided because the sub-expression to be evaluated is isolated at the top level of the state. This possibility is pursued in Assignment 8.

Note that the left-to-right ordering between processes is necessary to guarantee progress. An example which demonstrates this is the configuration

$$l_1{=}\mathtt{pair}(\mathtt{fst}(l_2), \mathtt{num}(3)), l_2{=}\mathtt{pair}(\mathtt{fst}(l_1), \mathtt{num}(4))$$

which is locally well-typed with l_1 and l_2 of type $\mathsf{int} \times \mathsf{int}$, but in which both process are blocked, waiting for each other. This situation is referred to as a *deadlock*.

The typing judgment on process configurations must take this into account. It has the form $H : \Lambda$, where Λ assigns types to processes. We also generalize the typing judgment for expressions to allow labels to occur—they are simply propagated except in the one rule shown below.

$$\frac{}{\cdot : \cdot} \qquad \frac{H : \Lambda \quad \Lambda; \cdot \vdash e : \tau}{(H, l{=}e) : (\Lambda, l{:}\tau)}$$

$$\frac{l{:}\tau \text{ in } \Lambda}{\Lambda; \Gamma \vdash l : \tau}$$

The preservation theorem is not difficult to formulate.

Theorem 1 (Preservation)

(i) If $H : \Lambda$ and $\Lambda; \cdot \vdash e : \tau$ and $\langle H, e\rangle \mapsto \langle H', e'\rangle$ then there is a $\Lambda' \supseteq \Lambda$ such that $H' : \Lambda'$ and $\Lambda'; \cdot \vdash e' : \tau$.

(ii) If $H : \Lambda$ and $H \mapsto H'$ then there is a $\Lambda' \supseteq \Lambda$ such that $H' : \Lambda'$.

Proof: By induction on the derivation of the step relation, applying inversion on the typing assumptions. ■

The progress theorem requires more care. We first formalize the notion of a terminal state.

$$\frac{}{\cdot \ \mathsf{terminal}} \qquad \frac{H \ \mathsf{terminal} \quad v \ \mathsf{value}}{(H, l{=}v) \ \mathsf{terminal}}$$

Theorem 2 (Progress)

(i) If $H_1 : \Lambda_1$, H_1 terminal, and $\Lambda_1; \cdot \vdash e : \tau$ then either

(a) e value, or

(b) there exists H_1' and e' such that $\langle H_1, e\rangle \mapsto \langle H_1', e'\rangle$

(ii) If $H : \Lambda$ then either

(a) H terminal, or

(b) there exists H' such that $H \mapsto H'$

Proof: For (i) by induction on the derivation of $\Lambda_1; \cdot \vdash e : \tau$, using a generalization of value inversion that permits labels. Labels must be defined in H_1 and bound to values (since H_1 is terminal), thereby assuring progress.

For (ii) by appeal to (i) given $H = H_1, l{=}e, H_2$, where e is not a value. Such a decomposition must be possible if H is not terminal. ■

We close this lecture with a two examples of programs written using the future construct. These have been adapted from Halstead's paper, but are present in ML assuming a construct `future(e)`. A simple sequential simulation is simply to define `future` as the identity function.

The first example is the insertion of a node into an ordered binary tree. An ordered binary tree is either `Empty`, a data-carrying `Leaf(x)`, or a node `Node(left,y,right)` where `y` is a discriminator so that every element in the left subtree `left` is smaller or equal to `y`, and every element in the right subtree `right` is larger than `y`.

The parallelism in this example is the possibility to spawn a thread at each recursive call to `insert`, which returns immediately and continues insertion of the subtree. Thereby, if we insert several elements in a row, the computations can ripple down the tree simultaneously almost in a pipeline structure (although there is no assumption that the operations are indeed performed in lock-step).

```
datatype Tree =
    Empty
  | Leaf of int
  | Node of Tree * int * Tree
fun insert (x, Empty) = Leaf(x)
  | insert (x, tree as Leaf(y)) =
    if y < x
      then Node (tree, y, Leaf(x))
    else Node (Leaf(x), x, tree)
  | insert (x, Node(left, y, right)) =
    if y < x
      then Node (left, y, future (insert (x, right)))
    else Node (future (insert (x, left), y, right))
```

As a second example, we consider `quicksort`, implemented on lists. It first partitions a list into elements smaller and greater than a pivot element (the first element in the list) and then sorts the sublists in parallel before appending them. There is also a smaller amount of parallelism in the `partition` function shown below.

```
fun quicksort (nil, acc) = acc
  | quicksort (x::l, acc) =
    let
      val (smaller, greater) = partition (x, l)
    in
      quicksort (smaller,
                 x::future (quicksort (greater, acc)))
    end
and partition (x, nil) = (nil, nil)
  | partition (x, y::l) =
    let
      val parts = future (partition (x, l))
    in
      if y < x
        then (y::future(#1(parts)), future (#2(parts)))
      else (future (#1(parts)), y::future (#2(parts)))
    end
```

Stating a preservation theorem for MinML with futures is not difficult. However, proving progress is tricky, because typing alone among multiple processes does not rule out the possibility of a deadlock. Instead, we must assume a partial order among processes so that minimal elements in the order cannot block. It is important that his order is maintained during computation and that it remains an order, that is, remains acyclic. Again, we will not pursue this direction further.