Assignment 8: Concurrent Programming with Futures

15-312: Foundations of Programming Languages
Daniel Spoonhower (`spoons+@cs.cmu.edu`)

Out: Thursday, November 13, 2003
Due: Thursday, December 4, 2003 (**11:59:59 pm**)

150 points total + possible extra credit

Introduction

In this assignment, you will construct an interpreter for a version of MinML extended with *futures*, instrument the interpreter to allow certain measurements to be taken, and run your interpreter on standard algorithms parallelized in different ways. You will then analyze the results.

You will rarely, if ever, need to write long or complicated functions to complete this assignment. Therefore, you should strive for elegance. Your solution will be graded primarily on correctness, but if your code does not correctly handle one or more cases, we will inspect your code and attempt to give you some credit for the understanding it reflects. In other words, it is to your benefit to write clean, legible code.

The state of MinML

For the sake of simplicity, we will revert to the style of evaluation found in Assignment 4; there will be no garbage collection in this assignment. Moreover, we have also made the following changes.

- Exceptions are no longer supported. The corresponding state $k \mid \eta \ll v$ has been removed.
- One particular recursive type has been added, a type `tree`, representing binary integer trees. This is essentially equivalent to the following SML datatype of binary trees with records at branch nodes:

```
datatype tree = Leaf
              | Node of int * tree * tree
```

Trees can be created in the way you would expect from using SML

```
Leaf      Node(3, Leaf, Node(2 + 2, Leaf, Leaf))
```

and so forth. Trees can be taken apart with `case`, as shown here:

```
case t of
   Leaf => e1
 | Node(x, l, r) => e2
```

The constructor of the `exp` datatype is `Case(e,e1,(x,l,r,e2))` where `x`, `l`, `r` are bindings.

It will come as no surprise (at least to Lisp and Scheme hackers) that this type does double duty as a list type, since one can represent a list containing $x_1, x_2, \ldots, x_n$ by a severely unbalanced ("rightist") tree:

```
 x1
/  \
   x2
  /  \
      .
       .
        .
         \
         xn
        /  \
```

Lists (again, unbalanced trees) may also be written using a form of syntactic sugar found in SML. For example, the following two examples are equivalent.

```
[1, 3]      Node(1, Leaf, Node(3, Leaf, Leaf))
```

- Programs can take input in the following sense, typing

  ```
  Top.apply_eval "foo.mml" e
  ```

  where `e` has type `DBMinML.exp` and `foo.mml` is a file containing a MinML program of the form

  ```
  fn x : t => b;
  ```

  will cause `e` (which should represent MinML expression of type `t`) to be evaluated and substituted for `x` in `b`. This comes in handy for the last part of this assignment. There is also an analogous function, `Top.apply_step`.

- We have added a new state to the E machine, represented by the SML datatype constructor Done(v), to explicitly denote machines that have finished computation. This new state appears in only one rule, shown below.

$$\bullet \mid \eta < v \quad \mapsto_{\mathsf{e}} \quad = v$$

Semantics of futures

The assignment hinges on an extension of the E machine formalism and interpreter (Assignment 4) to support futures.

Static semantics

A nice property of futures is that they do not complicate the type system. If e has type τ, $\texttt{future}(e)$ also has type τ. The typing rule is shown below.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{future}(e) : \tau}\ \textit{FutureTyp}$$

Informal dynamic semantics

As discussed in lecture, the expression `future(e)` returns immediately with a placeholder value, called a *promise*. Evaluation of e begins immediately, in parallel, while the original thread continues independently. The promise is a value, and the original thread will block waiting for the new thread only when (and if) the promise is *touched*—that is, only when a (non-promise) value is actually required. For example, the promise returned by `future(3 + 5)` in the expression

```
future(3 + 5) * 2
```

will be touched almost immediately, because the primop `*` needs the actual values of both arguments to compute the product. Of course, the new thread may have computed the value of e by that time, in which case the original thread immediately retrieves the computed value and continues.

As another example, suppose `f` is a function returning a tree.

```
let t = future(f(e)) in
  case t of Leaf => ...
          | Node(x, l, r) => ...
end
```

Evaluating the above `case` expression touches the promise returned by `future(f(e))`, since we cannot know which branch of the `case` to evaluate unless we know what `t` is.

Keep in mind, however, that promises are values. Thus, despite remaining in a call-by-value setting, simply passing a promise to a function, constructing a pair in which one component is a promise, or constructing a `Node` in which one or more subtrees are promises, does *not* touch the promise. In this sense, futures are similar to the `delay` construct seen in Assignment 4.

The parallel E machine

To model the various threads spawned by use of futures, we extend the E machine of Assignment 4 as follows. Instead of a single state s of the form $k \mid \eta > e$

or $k \mid \eta < v$ where k is a stack and η an environment, we have a set P of threads, each of which we represent by a *thread identifier* p (which in turn is represented by an integer in the implementation) and a state. The general picture is

$$p_1 : k_1 \mid \eta_1 > e_1, \qquad p_2 : k_2 \mid \eta_2 < v_2, \qquad \ldots, \qquad p_n : k_n \mid \eta_n > e_n$$

(where we could equally well have seen $k_1 \mid \eta_1 < v_1$ for some v_1). When an expression `future(e)` is evaluated, it should return a promise and spawn a new thread. This new thread should begin evaluating e in parallel and with respect to an empty stack.

Task: Semantics of Futures (30 points)

Write a set of transition rules for MinML with futures in the above style. You need not write rules that are unchanged (except for the addition of the thread set P) from a language without futures, such as

$$P,\ \ k \mid \eta > \texttt{Int}(k) \quad \mapsto_{\mathsf{e}} \quad P,\ \ k \mid \eta < \texttt{Int}(k)$$

To represent the state of a thread with stack k and environment η blocking on a promised value, a value being computed by a spawned thread p', write

$$k \mid \eta\ \mathsf{blocked}(p')$$

This notation is only a suggestion; if you need to maintain more information in a blocked thread (besides p'), you are welcome to do so. In fact, you may need to extend the possible states in other ways. Along with the rules, submit a grammar for states s. Our proposed grammar so far would be written

$$\begin{aligned} s ::= {} & k \mid \eta > e \\ \mid {} & k \mid \eta < v \\ \mid {} & k \mid \eta\ \mathsf{blocked}(p) \\ \mid {} & = v \end{aligned}$$

Important: It's useful to have a correct semantics *before* you start implementing it. You are strongly encouraged to complete this question quickly and send me your proposed semantics. I will read it; if it looks right I will tell you that, if there are problems I will point them out. If possible, send me your rules by November 21 (but I can read them at any time). This is provided as a public service; please take advantage of it!

Your final hand-in should be on paper, or as a text, PostScript, or PDF file called `rules.txt` (`rules.ps`, `rules.pdf`) in your handin directory.

Task: Implementing Futures (60 points)

In this task, you will implement your rules in `e-mach.sml`. First, however, you should extend the datatype `state` so that it corresponds to your grammar for states.

Second, revise the outer part of the implementation (everything besides `step`) so that, instead of a single state, a set of states—along with integer thread identifiers—is maintained. At this point, of course, you haven't implemented evaluation of `future(e)`; this is preparation for the next step.

Finally, implement your rules. The relevant MinML abstract syntax constructors are `Future of exp` and `Promise of int`; you can change the latter to something other than `int` if necessary.

Imagine you have an unlimited number of processors available, so a processor can be dedicated to each thread. Scheduling is almost beside the point with this assumption; you can just go through the list of threads in round-robin fashion and try to step each one.

One final note, you are *not* required to maintain the code in `stepStream` and `evaluateStream`. You may find it to be useful to update this code, but you will not be graded on your implementation. If you choose not to maintain it, I suggest raising an exception.

Experimenting with Futures

Once you have a working interpreter, you can start experimenting. We're interested in the performance increase (or decrease!) when a sequential program is changed to use futures and run on a multi-processor.

Since the interpreter itself is a sequential program, what can we actually measure? The running time of the interpreter doesn't tell us much. To get some interesting (though excessively optimistic) measurements, you need to instrument your interpreter to collect some statistics. In particular, it should measure the following:

1. *clock ticks:* the number of times you looped through the set of threads
2. *work:* the number of times a step was taken

Just before returning the value that is the result of evaluation, the function `evaluate` in `e-mach.sml` should print the statistics, labeled appropriately. As mentioned above, you can use `Top.apply_eval` to pass in a "prefabricated" input to the MinML program you are experimented with.

Task: Experiments (60 points)

First, gather statistics for `filter.mml` and `filterf.mml` (a version with futures added), with the input as the *complete* and *rightist* trees returned by the functions `Top.complete` and `Top.seq`. These MinML programs filter through the nodes of a tree producing a list of integers that satisfy some given condition. Run complete trees through level 8 (`Top.complete 2` through `Top.complete(8)`) and rightist trees at all powers of 2 from 2^2 to 2^8.

Next, implement quicksort in MinML (without futures). Then add futures in whatever way seems most likely to be effective. Compare them on input

```
Top.random_seq (17,27) n
```

for n from 10 to 100.

Finally, implement sets represented as ordered trees in MinML, with insert, union, and intersection operations. Again, you should implement two versions: one without futures and one version with futures. You may find it useful to write an SML version first. Try the union and intersection of:

1. two rightist trees (`Top.seq`) of the same size, of sizes from 2^2 to 2^8 (powers of 2);
2. two random trees (`Top.ordered_random seed n`) of size n, for all n up to 64. Use a different seed for each tree!

Turn in the algorithms you implement as files `qsort.mml`, `qsortf.mml` (with futures), `sets.mml`, and `setsf.mml` (with futures) in the handin directory.

Write a report on your results; graphs are strongly encouraged. Discuss and explain the results. If you don't fully understand the behavior, analyze possible explanations. Turn in your report on paper or as a PostScript (`report.ps`) or PDF file (`report.pdf`).

If your interpreter is too slow to run all the cases in a reasonable amount of time, just run fewer cases (for example, you could take only every tenth n from 10 to 100).

A note on grading

This task is worth 60 points. Partial credit will be given for any or all of the following:

- instrumenting your interpreter
- implementing quicksort in MinML
- implementing set functions in MinML
- adding futures to either or both
- measuring and handing in raw numbers without commenting on them (but of course you can't get full credit unless you carefully analyze and discuss your results)

Exceptionally insightful work will lead to extra credit. Collecting statistics not listed above (for variously shaped trees, or adding futures in more than one way), and reporting on them, can get extra credit as well.

Task: Scheduling (15 points EXTRA CREDIT)

One reason the simulation is unrealistic is that the number of processors is assumed to be infinite, allowing hundreds of threads to run simultaneously. For extra credit, revise your interpreter to schedule threads for some number k (where k is a parameter to the interpreter) of processors. Turn in any additional files you need to modify for this task.

Hand-in Instructions

Turn in the rules on paper, or as a text, PostScript, or PDF file called `rules.txt` (`rules.ps`, `rules.pdf`) in the handin directory

`/afs/andrew/scs/cs/15-312/students/`*Andrew user ID*`/asst8/`

by 11:59 pm on the due date. Immediately after the deadline, we will run a script to sweep through all the handin directories and copy your files elsewhere. We will also sweep 24, 48, and 72 hours after the deadline, for anyone using late days on this assignment.

You should also turn in `e-mach.sml`, `qsort.mml`, `qsortf.mml`, `sets.mml`, and `setsf.mml` (and any other files you modify, *e.g.* `print.sml`) by copying them to your handin directory. Finally, turn in the report on paper, or as a PostScript (`report.ps`) or PDF file (`report.pdf`) again, in the handin directory.

If you decide to turn in answers on paper, your solution is due in WeH 5119 by 11:59 pm on the due date. If you are handing the assignment in late, either submit your solution electronically or bring it to WeH 5119. If I am not in my office, write the date and time at the top and leave it on my chair or (if the door is closed) slide it underneath the door.

NOTICE: This is the last assignment, so you're welcome to use up your remaining late days. Please look at your grades on Blackboard to make sure you really do have late days left! You cannot use more late days than you have left. If you hand in 3 days (more than 72 hours) late, and you had fewer than 3 days left, **your assignment will not be graded**. Likewise, if you have no days left, you **must** hand in on time (11:59 pm, December 4) to receive a grade.

For more information on handing in code, refer to

`http://www.cs.cmu.edu/~fp/courses/312/assignments.html`