Assignment 8:
Concurrent Programming with Futures

15-312: Foundations of Programming Languages
Joshua Dunfield (joshuad@cs.cmu.edu)

Out: Thursday, November 14, 2002
Due: Thursday, December 5 (**11:59:59 pm**)

150 points total + possible extra credit

Revised December 2, 2002

Introduction

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

Note: In the .sml files, most changes from Assignment 6 are indicated like this:

```
(* new asst8 code: *)
...
(* end asst8 code *)
```

The state of MinML

Some things have become obsolete (and have either been removed or left in a semi-supported state), in which case you need not worry about them. Some things have been added. This section summarizes the changes from previous assignments.

- A non-recursive function construct `lam x in e end` has been added.
- Programs can take input, in the following sense: typing

```
Top.evaluate "foo.mml" e
```

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

```
lam arg in
  body
end : tau1 -> tau2
;
```

will cause `e` (which should be a MinML value of type `tau1`) to be substituted for `arg` in `body` before evaluation. This comes in handy for the last part of this assignment.

- As in Assignment 6, exceptions and continuations are not supported.
- Floats are not supported. Since this eliminates most of the utility of subtyping, subtyping is not supported either.
- Existential types are not supported.
- Sums are not supported, but one particular recursive type has been added: a type `tau tree` for any type `tau`. This is essentially equivalent to the following SML datatype of binary trees with records at branch nodes:

```
datatype 'a tree = Leaf
                 | Node of 'a * 'a tree * 'a tree
```

It is polymorphic, so you can build trees containing integers, pairs of integers, and so on.

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

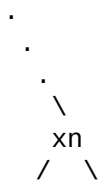
```
Leaf      Node(3, Leaf, Node(2 + 2, Leaf, Leaf))
```

and so forth. Trees can be taken apart with `case`:

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

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
  /  \
```



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

- Typing is still done bidirectionally, so there are various type annotations floating around the code. The function `erase` in `c-mach.sml` removes the type annotations so they don't get in the way during evaluation.
- Some other new constructs are discussed below.

Semantics of futures

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

Futures: static semantics. A nice property of futures is that they do not complicate the type system. If e has type τ, `future`(e) also has type τ. So the bidirectional typing rules are just

$$\frac{\Gamma \vdash e \uparrow \tau}{\Gamma \vdash \texttt{future}(e) \uparrow \tau}\ (\textit{FutureUp}) \qquad \frac{\Gamma \vdash e \downarrow \tau}{\Gamma \vdash \texttt{future}(e) \downarrow \tau}\ (\textit{FutureDown})$$

Futures: informal dynamic semantics. As discussed in lecture, the expression `future(e)` returns immediately with a future, here called a *promise*. Evaluation of e begins immediately, in parallel, in a new thread; the original thread continues. The promise is a value, and the original thread will block waiting for the new thread only when (and if) the promise is *touched*—which happens when a 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 difference. 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 of some kind.

```
let t = future(f(e))
in
  case t of Leaf => ... | Node(x,l,r) => ... end
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, or constructing a pair in which one component is a promise, or constructing a `Node` in which both subtrees are promises, does *not* touch the promise.

The parallel C-machine. To model the various threads spawned by use of futures, we extend the C-machine of Assignment 4 as follows. Instead of a single state s of the form $k > e$ or $k < v$ where k is a stack, we have a set P of threads, each of which we can represent by a *thread identifier* p (represented by an integer in the implementation) and a state. The general picture is

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

(where we could equally well have $k_1 < v_1$ for some v_1, etc.) When an expression `future(e)` is evaluated, it should return a promise and spawn a new thread:

$$P,\ \ p : k > \texttt{future}(e) \ \mapsto_{\mathsf{c}}\ P,\ \ p : k < \texttt{promise}(p'),\ \ p' : \bullet > e$$

where $\bullet$ is the empty stack and p' is a fresh thread identifier. The rule leaves the other threads P alone, returns a new promise to k, and begins evaluating e with respect to an empty stack in the new thread p'.

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

$$P,\ \ k > \mathsf{num}(\overline{k}) \quad \mapsto_{\mathsf{c}} \quad P,\ \ k < \mathsf{num}(\overline{k})$$

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

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

This notation, and the rule given above for evaluation of `future(e)`, are only suggestions; if—for whatever reason—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

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

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[1] me your rules by November 22 (but I can read them at any time). This is provided as a public service; please take advantage of it!

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)

Take your rules and implement them in `c-mach.sml`. First, write down a datatype `state` corresponding to your grammar for states. Then modify the existing `eval` function to return a state instead of a value. For example, when evaluating `If`(e, e_1, e_2), `eval` should actually return the state $k \triangleright \mathsf{if}(\Box, e_1, e_2) > e$. You can continue to represent the stack k by an ML function (you may also encode the stack by a datatype if you really want to).

Second, revise the outer part of the implementation (everything besides `eval`) 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 absolutely necessary, but I don't know why you'd want to.

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.

Experimenting with Futures (60 points)

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

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 collect 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.

[1] A text e-mail is fine; you can also give me the rules on paper, or use your handin directory as a drop-box for text, PostScript, or PDF files. I prefer *not* to receive e-mail attachments.

Just before returning the value the program evaluated to, the function `evaluate` in `c-mach.sml` should print the statistics, labeled appropriately.

As mentioned above, you can use `Top.evaluate` to pass in a "prefabricated" input to the MinML program being experimented with.

First, gather statistics for `collect.mml` and `collectf.mml` (a version with futures added), with the input as the *complete* and *rightist* trees returned by the functions `Top.complete` and `Top.seq`. 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. 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.

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).

Grading: This part 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
- collecting 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.

Extra credit: scheduling. 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.

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.

Turn in `c-mach.sml`, `qsort.mml`, `qsortf.mml`, `sets.mml`, and `setsf.mml` by copying them to your handin directory.

Turn in the report on paper, or as a PostScript or PDF file `report.ps`, `report.pdf` in the handin directory.

WARNING: MAKE SURE YOUR TABS ARE SET TO 8 SPACES, or replace tabs with spaces before submitting. I am sick of reading code with messed-up indentation. This does *not* mean you have to indent by 8 spaces; indeed, the point of this is to keep the width down to something reasonable, so please indent by less than 8 spaces.

Turn in non-programming questions as text, PostScript, or PDF files in the handin directory. Or, if you wish, you may turn in answers on paper, due in WeH 1313 by 11:59 pm on December 5. **If you are using late days, paper handin is by arrangement only** (send mail and we'll figure something out).

NOTICE: This is the last assignment, so you're welcome to use up your remaining late days. But 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 5) to receive a grade.

For more information on handing in code, refer to

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