# EXTRA CREDIT 2 — IMPLEMENTING MACHINES

COMP 3010 — ORGANIZATION OF PROGRAMMING LANGUAGES

## 1. Functions and Primitives

In assignment 10, you implemented an interpreter for this programming language based on the $\lambda$-calculus with primitive constants (numbers and booleans) and operations:

$$\begin{aligned} n &::= 0 \mid 1 \mid 2 \mid 3 \mid \ldots \\ b &::= \mathit{true} \mid \mathit{false} \\ x &::= \text{any symbolic name} \\ M, C &::= n \mid b \mid x \mid C\ C \mid \lambda x{:}T.C \mid \texttt{if}\, C\, \texttt{then}\, C\, \texttt{else}\, C \\ T &::= \texttt{num} \mid \texttt{bool} \mid T \to T \end{aligned}$$

The interpreter for assignment 10 was based on the *big-step semantics* for the programming language, which is implemented as a *recursive* function that calls itself to evaluate any sub-expressions needed to determine the answer to a particular syntax tree. Part of this implementation involved interpreted functions as first-class values by representing them as *closures*, which pair together a function (which includes both the function parameter and the body of code to run when it's called) along with the environment recording the values stored in any free variables that were in scope at the time the closure was created.

These extra credit exercises ask you to write a more efficient implementation of this same programming language in terms of an *abstract machine*, and then to extend that language with additional features that can be more easily expressed in the machine.

Recall the CEK machine from class, which abstractly models the steps of a real machine in terms of two configurations

- $\mathit{Eval}\langle C \mid E \mid K\rangle$ or
- $\mathit{Ret}\langle V \mid K\rangle$

where $E$ stands for an *Environment*, $K$ stands for a *Kontinuation stacK* made up as a sequence of *Frames* $F$, and $V$ stands for a *Value* defined by the following grammar:

$$\begin{aligned} E &::= \epsilon \mid x = V, E \\ F &::= \square\ [E]C \mid V\ \square \\ K &::= \texttt{done} \mid F; K \\ V &::= n \mid b \mid [E]\lambda x.C \end{aligned}$$

Loading an *initial configuration* $\mathit{Eval}\langle C \mid \epsilon \mid \texttt{done}\rangle$ lets you evaluate the *Code* $C$ to get its answer, which is found in the value of a *final configuration* of the form

*Date*: Summer 2022.

$Ret\langle V \mid \texttt{done}\rangle$. To go from the initial to final configuration, the machine iteratively applies one of these transition rules, one after the other, until the answer is reached:

$$\begin{aligned}
Eval\langle C\ C' \mid E \mid K\rangle &\mapsto Eval\langle C \mid E \mid \square[E]C; K\rangle \\
Eval\langle x \mid E \mid K\rangle &\mapsto Ret\langle V \mid K\rangle && (x = V \in E) \\
Eval\langle n \mid E \mid K\rangle &\mapsto Ret\langle n \mid K\rangle \\
Eval\langle b \mid E \mid K\rangle &\mapsto Ret\langle b \mid K\rangle \\
Eval\langle \lambda x.C \mid E \mid K\rangle &\mapsto Ret\langle [E]\lambda x.C \mid K\rangle
\end{aligned}$$

$$\begin{aligned}
Ret\langle V \mid \square\ [E]C; K\rangle &\mapsto Ret\langle C \mid E \mid V\ \square; K\rangle \\
Ret\langle V \mid ([E]\lambda x.C)\ \square; K\rangle &\mapsto Eval\langle C \mid x = V, E \mid K\rangle
\end{aligned}$$

**Extra Credit Exercise 1.** Write a *non-recursive* function that implements the CEK machine as described above. In particular, your function should

(1) take the source code $C$ to evaluate,
(2) create an initial machine configuration $Eval\langle C \mid \epsilon \mid \texttt{done}\rangle$ as the starting state of the machine,
(3) implement the main engine of the machine in a (while) loop, that repeatedly changes the current state of the machine according to the transition rules above by
    (a) matching current state of the machine against the left-hand side of $\mapsto$ in one of the rules, and then
    (b) updating the state of the machine ad described by the right-hand side of $\mapsto$ in that same matching rule, and then
    (c) repeating the loop again from the top
(4) the (while) loop engine comes to a stop when the current machine state matches a final configuration $Ret\langle V \mid \texttt{done}\rangle$. When you reach this state, end the loop by returning the value $V$ as the final answer of the program.

*Hint:* The environment $E$ can be represented as an ordinary hash map from variable names to machine values $V$, similar to assignment 10. The continuation $K$ can be represented as a stack/list/array, where `done` is the empty list, and pushing a frame onto a continuation $F; K$ (like $V\ \square; K$) is represented by adding a representation of the frame $F$ (implementing $V\ \square$ and $\square\ [E]C$ as concrete data structures $\texttt{apply}(V)$ and $\texttt{arg}(E, C)$) to the front of the stack.

**Extra Credit Exercise 2.** Extend your implementation of the CEK machine from extra credit exercise 1 with primitive operations. These primitive operations are added to the language similarly to assignment 10:

$$V ::= \cdots \mid p\#n\ V_1 \ldots V_i \qquad (i < n)$$

along with an implementation of the function $prim(p\#n, V_1, \ldots, V_n)$ implemented for every primitive operation $p$ of arity $n$ with $n$ arguments. To start with, you can implement every one of the primitive operations from assignment 10, and consider adding additional useful primitive operations on numbers and booleans that you can think of.

To add primitive operations to your CEK machine, you will need a representation of partially-applied operations (written above as $p\#n\ V_1 \ldots V_i$ when the number of

arguments $i$ is less than $p$'s expected parameters $n$). You will also need to implement the extra transition rules that handle the cases when primitive operations are partially or fully applied:

$$\mathit{Ret}\langle V \mid p\#i{+}1\ V_1 \ldots V_i\ \square; K\rangle \mapsto \mathit{Ret}\langle V' \mid K\rangle \qquad (\mathit{prim}(p\#i{+}1, V_1, \ldots, V_i, V) = V')$$
$$\mathit{Ret}\langle V \mid p\#n\ V_1 \ldots V_i\ \square; K\rangle \mapsto \mathit{Ret}\langle p\#n\ V_1 \ldots V_i\ V \mid K\rangle \qquad (i+1 < n)$$

Finally, define a starting environment which binds a variable in the language to a value representing each primitive operation you implement in *prim*. This starting environment should be used instead of the empty one ($\epsilon$) in the initial configuration of your machine implemented in extra credit exercise 1. Again, you can look to assignment 10 for inspiration on who this is done.

## 2. References and Memory

These exercises ask you to extend your abstract machine with a store, that can remember the current state of updatable values in dynamic memory.

**Extra Credit Exercise 3.** Implement stores as a data structure. Stores can be represented as essentially the same thing as hash maps — with locations used as keys that are mapped to machine values — but with an extra operation (*fresh*) that generates a new location key that is not associated with anything in the store yet.

Then extend your CEK machine from section 1 with a store $S$,

$$S ::= \epsilon \mid l := V, S$$

by generalizing the two configurations to

- $\mathit{Eval}\langle C \mid E \mid S \mid K\rangle$
- $\mathit{Ret}\langle C \mid S \mid K\rangle$

This also means that you will have to begin your initial configuration with the empty store $\epsilon$, and also you will have to update your implementation of every transition rule to expect a store $S$ on the left-hand side which is copied over to a similar store $S$ on the right-hand side.

**Extra Credit Exercise 4.** Add additional primitive operations (following extra credit exercise 2) that allow programs to access and use references into the store. In particular, you can add three primitive operations:

- `ref`#1 takes one argument (which could be any value), allocates an empty reference cell in a *fresh* location in the current store, copes its argument into that location, and returns the address of the location.
- `read`#1 takes one argument (which *must* be a location), and reads the value currently stored at that location in the store, and returns it.
- `write`#2 takes two arguments (the first *must* be a location, and the second can be any value), and overwrites the store so that the location given by the first argument now contains the value of the second argument.

More formally, these three operations can be implemented by extending the *prim* function (which says what to do for each primitive operation) by taking the current state of the store as an argument, returning a (potentially) new state of the store as a result along with the value, and adding these additional cases for the stateful

operations above:

$$\begin{aligned} \mathit{prim}(\texttt{ref\#1}, S, V) &= l, (l := V, S) && (l \text{ fresh in } S) \\ \mathit{prim}(\texttt{read\#1}, S, l) &= V, S && (l := V \in S) \\ \mathit{prim}(\texttt{write\#2}, S, l, V) &= V, (l := V, S) \end{aligned}$$

All other primitive operations from before in extra credit exercise 2 should be generalized so that they do not interact with the store $S$ (they return the same store that they were given, and the value they return is the same no matter what is in the store). For example,

$$\mathit{prim}(\texttt{plus\#2}, S, n_1, n_2) = (n_1 + n_2), S$$

To handle primitive operations that might refer to, or modify, the store, you will also need to generalize the transition rule implementing the case where a primitive operation is run, following this specification:

$$\mathit{Ret}\langle V \mid S \mid p\#i{+}1\ V_1 \ldots V_i\ \square; K\rangle \mapsto \mathit{Ret}\langle V' \mid S' \mid K\rangle \quad (\mathit{prim}(p\#i{+}1, S, V_1, \ldots, V_i, V) = V', S')$$

Note that the only change for partially applying a primitive operation is to carry the current state of the store from the left-to-right-hand sides of the rule, but otherwise doesn't change.

## 3. Control and Recovery

These exercises ask you to implement control structures that let you traverse or remember the continuation stack. In the case where you did not implement the store component of the machine yet from section 2, you can ignore the $S$ component in the following descriptions.

**Extra Credit Exercise 5.** Extend your language of source programs to allow for throwing and catching exceptions. This involves adding the new cases for expressions and continuation frames in the grammar of syntax:

$$\begin{aligned} M, C &::= \cdots \mid \texttt{throw}\, T\ C \mid \texttt{try}\, C\, \texttt{catch}\ (T\ x)\ C \\ F &::= \cdots \mid \texttt{throw}\, T\ \square \mid \texttt{try}\, \square\, \texttt{catch}\ (T\ x)\ [E]C \end{aligned}$$

It also involves adding the new form of machine configuration

$$\mathit{Throw}\langle T \mid V \mid S \mid K\rangle$$

for throwing the exceptional value $V$ of type $T$ up the stack $K$.

The extra transitions you will have to implement in your machine interpreter should follow these specification rules:

$$\begin{aligned} \mathit{Eval}\langle \texttt{throw}\, T\ C \mid E \mid S \mid K\rangle &\mapsto \mathit{Eval}\langle C \mid E \mid S \mid \texttt{throw}\, T\ \square; K\rangle \\ \mathit{Eval}\langle \texttt{try}\, C\, \texttt{catch}\ (T\ x)\ C' \mid E \mid S \mid K\rangle &\mapsto \mathit{Eval}\langle C \mid E \mid S \mid \texttt{try}\, \square\, \texttt{catch}\ (T\ x)\ [E]C'; K\rangle \end{aligned}$$

$$\begin{aligned} \mathit{Ret}\langle V \mid S \mid \texttt{throw}\, T\ \square; K\rangle &\mapsto \mathit{Throw}\langle T \mid V \mid S \mid K\rangle \\ \mathit{Ret}\langle V \mid S \mid \texttt{try}\, \square\, \texttt{catch}\ (T\ x)\ [E]C; K\rangle &\mapsto \mathit{Ret}\langle V \mid S \mid K\rangle \end{aligned}$$

$$\begin{aligned} \mathit{Throw}\langle T \mid V \mid S \mid \texttt{try}\, \square\, \texttt{catch}\ (T\ x)\ [E]C; K\rangle &\mapsto \mathit{Eval}\langle C \mid x = V, E \mid S \mid K\rangle \\ \mathit{Throw}\langle T \mid V \mid S \mid \texttt{try}\, \square\, \texttt{catch}\ (T'\ x)\ [E]C; K\rangle &\mapsto \mathit{Throw}\langle T \mid V \mid S \mid K\rangle && (T \neq T') \\ \mathit{Throw}\langle T \mid V \mid S \mid F; K\rangle &\mapsto \mathit{Throw}\langle T \mid V \mid S \mid K\rangle && (\text{otherwise}) \end{aligned}$$

**Extra Credit Exercise 6.** Extend your language with first-class continuations captured by the primitive operation `call/cc`. This involves extending the syntax of values to include captured continuations:

$$V ::= \cdots \mid \texttt{cont}\, K$$

that act like functions. When applied to an argument, continuations should take over the current call stack according to this following specification rule:

$$\mathit{Ret}\langle V \mid S \mid \texttt{cont}\, K'\ \square; K\rangle \mapsto \mathit{Ret}\langle V \mid S \mid K'\rangle$$

To generate one of these first-class continuation values, add a new primitive operation (of arity 1) named `call/cc` which takes a snapshot of the current state of the continuation stack $K$ and stores it inside of an immutable value $\texttt{cont}\, K$. To do so, you can further extend the rule for running primitive operations to also be given a reference to the call stack like so:

$$\mathit{Ret}\langle V \mid S \mid p\#i{+}1\ V_1 \ldots V_i\ \square; K\rangle \mapsto \mathit{Ret}\langle V' \mid S' \mid K'\rangle \quad (\mathit{prim}(p\#i{+}1, S, K, V_1, \ldots, V_i, V) = V', S', K')$$

That way, the primitive `call/cc` operator can be implemented according to this specification

$$\mathit{prim}(\texttt{call/cc}\,\#1, S, K, V) = (\texttt{cont}\, K), S, (V\ \square; K)$$

so that the generated $\texttt{cont}\, K$ which copies the current continuation stack is passed as an argument to the function $V$. For example, evaluating code $\mathit{plus}\ 1\ (\mathit{call/cc}(\lambda k.k\ 3))$ — in an initially empty continuation and an initial environment where *plus* is bound to a primitive operation `plus`#2 and *call/cc* is bound to `call/cc` #1 described above — should eventually call the function $\lambda k.k\ 3$ with the argument `cont` (`plus`#2 1 $\square$; `done`).

*Hint:* `call/cc` should remember the current continuation stack in a *frozen* state. This means that the call stack stored in the captured continuation should always do the same thing, no matter how many times its used or how the future call stack changes. If you are implementing continuation stacks as mutable arrays, be very careful to make sure that the stack you capture will not be affected by future modifications of the call stack, for example, by copying the current call stack into a separate array that will be immutable, and then later copying it back into the current continuation when it is used.