An introduction to operational semantics and abstract machines

Hayo Thielecke
University of Birmingham, UK
http://www.cs.bham.ac.uk/~hxt/

February 29, 2012

Abstract

These notes are intended for a third-year course on Principles of Programming Languages. Their aim is to show how formalisms that originated in the lambda calculus can be extended and adapted to realistic programming language features. We use operational semantics and abstract machines to formalize a variety of programming language constructs, such as control and state. Relevance to research in programming languages will be pointed out on occasion.

Contents

1 Introduction

We start from the problem of how to evaluate closed terms in the lambda calculus in an efficient and realistic way. Familiarity with the basic notions of lambda calculus is assumed. This questions leads us to implementations of the core of a programming language using abstract machines.

Abstract machines are one of the canonical tools for studying programming languages. They can be seen as a particularly detailed form of operational semantics, in which things like bindings and context are made more explicit than in the lambda calculus and represented as data structures. Abstract machines are also similar to some models of computation, such as Turing Machines and Push Down Automata. In the same vein, the LL and LR parsers studied in compiler construction can be regarded as specialised abstract machines where grammar symbols are pushed onto and popped off the parsing stack.

Abstract machines are most closely associated with functional programming languages (such as Haskell, ML, or Scheme), but they can be adapted and extended for many programming language constructs. For instance, one could ask whether the (fairly complicated) type system of Java really succeeds in preventing all type errors at runtime. One way to do so is to define an abstract machine for running Java programs and then prove that types are preserved by all machine steps (in some appropriate sense). Another, and very different, example is verifying assembly code: in that case, one could build on an abstract machine that idealises the x86 architecture.

Abstract machines are not to be confused with Virtual Machines such as the JVM. The former are a research tools and the latter an implementation technique. Nevertheless, there are some similarities, as the JVM has a stack, much like many abstract machines. One could even start from an abstract machine and implement it as a virtual machine.

Writing an operational semantics is in many way like programming. More precisely, it is like programming an interpreter for the language at hand in an abstract mathematical language.

We will see two different ways to investigate programming language constructs via the lambda calculus. One is to encode some feature into the lambda calculus. We will do so for pairs. The other is to add new rules to a semantics for lambda calculus. We will do so for assignment and control.

An important theme in programming language theory is that we proceed by induction over syntax. Intuitively, the meaning of a program is given by taking it apart and combining the meaning of the pieces. Compiler writers call it "syntax-directed", logicians "compositionality".

Lambda expressions and closures are slowly making their way even into Java; see `http://openjdk.java.net/projects/lambda/`.

1.1 Lambda calculus

We assume familiarity with the lambda calculus. We write its syntax as follows:

$$\begin{array}{rcl} M & ::= & x \\ & | & (M_1\, M_2) \\ & | & \lambda x.M \\ & | & n \end{array}$$

Here n ranges over constants such as integers.

In an application

$$(M_1\, M_2)$$

we refer to the first term M_1 as the operator or function, and the second term M_2 as the operand or argument. We omit the parentheses in an application when they are clear from the context. Similarly, we write

$$M_1\, M_2\, M_3$$

as a shorthand for

$$((M_1\, M_2)\, M_3)$$

A term of the form

$$\lambda x.M$$

is called an abstraction, and x is called the bound variable.

We write $M_1[x \mapsto M_2]$ for the substitution in M_1 of x by M_2. Here we assume that the bound variables in M_1 do not occur in M_2, which can always be made to hold by renaming them.

Much of operational semantics can be seen as a generalization and refinement of β-reduction:

$$((\lambda x.M_1)\, M_2) \to_\beta (M_1[x \mapsto M_2])$$

The way beta reduction is done in implementations of programming languages is quite different from the presentation in the lambda calculus with renaming of bound variables, as literally implementing substitution would be far too inefficient.

1.2 Encoding of pairs

Data structures such as pairs, lists and trees can be encoded in the lambda calculus. The idea is a kind of protocol between a server that holds data and clients that need the data.

$$\begin{aligned} \langle M_1, M_2\rangle &= \lambda v.v\, M_1\, M_2 \\ \texttt{fst} &= \lambda p.p\,(\lambda x.\lambda y.x) \\ \texttt{snd} &= \lambda p.p\,(\lambda x.\lambda y.y) \end{aligned}$$

Exercise 1.1 (Easy) Prove that

$$\begin{aligned} \texttt{fst}\ \langle M_1, M_2\rangle &\to_\beta M_1 \\ \texttt{snd}\ \langle M_1, M_2\rangle &\to_\beta M_2 \end{aligned}$$

These encodings are closely related to the Visitor Pattern in object-oriented languages.

1.3 Finite partial functions

We write

$$\texttt{lookup}\, x\, \texttt{in}\, E$$

for the value we obtain by applying the partial function E to x.

Intuitively, a finite partial function is much like the `Map` interface in Java with its `put` and `get` methods:

```
V put(K key, V value);
V get(Object key);
```

Given a function E, the application `lookup` x `in` E corresponds to `E.get(x)` and the function extension

$$E[x \mapsto V]$$

is like `E.put(x, V)`.

Finite partial functions are used for a variety of different purposes in operational semantics: for simulating substitutions, environments, and heaps.

2 Run-time versus compile-time beta reduction

Beta reduction at runtime corresponds to function call. Beta reduction can also happen at compile time, as an optimization. One such optimization is function inlining. As an example, here is a snippet of C code:

```
int f(int x) { return x + 2000; }

int g(int y) { return f(1000); }
```

If we compile this code with optimization enabled, we can see that the function call in the body of `g` has been eliminated and replaced by the constant 3000. The compiled code is the same as if `g` had been defined like this:

```
int g(int y) { return 3000; }
```

In other words, the C compiler has performed two beta reductions.

We can observe that optimization by calling the compiler as `gcc -O4 -S` and reading the resulting assembly code. By contrast, with a lower level of otimization, the actual parameter 1000 is passed and `f` is called.

In the lambda calculus, the compiler optimzation is analogous to reducing the following expression to $\lambda y.3000$:

$$(\lambda f.(\lambda y.(f\,1000)))\,(\lambda x.(x + 2000))$$

Here $(\lambda y.(f\,1000))$ is analogous to `g` before optimization, $\lambda x.(x+2000)$ is analogous to `f`, and the abstraction over f models the fact that the definition of `f` is in scope in the body of `g`.

It seems fairly obvious that the optimized version of `g` that just returns a constant is more efficient. It also looks plausible that the optimized code behaves the same as the original code (at least as far as a user can tell without a debugger). But to justify program transformations like the above, we need some theory of what a program means, or does.

3 Big-step operational semantics

In the semantics of a programming language, we want every program accepted by the compiler to have a well-defined result. For some languages, like ML and Haskell, that is the case; for C not so much.

With beta reduction, there is no fixed evaluation strategy. As givem in Figure 1, reduction may happen anywhere in a term.

Consider the following term with multiple β-redexes R_1, R_2, and R_3:

$$\overbrace{(\lambda x_1 . (\underbrace{(\lambda x_2 . M_2)\, M_3}_{R_2}))\ (\underbrace{(\lambda x_4 . M_4)\, M_5}_{R_3})}^{R_1}$$

Under call-by-name, R_1 is the next redex to be reduced. Under call-by-value, it is R_3. R_2 is under a lambda.

$$(\lambda y.42)((\lambda x.xx)(\lambda x.xx))$$

The same term may produce a constant or loop forever, depending where we choose to perform the beta reduction.

We will use operational semantics do specify precisely what the evaluation order is. Under call-by-name, the above term evaluates to 42, whereas under call-by-value, it loops forever.

Big-step operational semantics tells us what the final result of a program is. Unlike abstract machines, a big-step semantics does not go into a lot of detail of describing the intermediate steps in running the programs.

We define values V as constants, variables, or λ abstractions.

$$\begin{array}{rcl} V & ::= & n \\ & | & x \\ & | & \lambda x.M \end{array}$$

In a big-step operational semantics, we define a relation between terms M and values V, written as

$$M \Downarrow V$$

Naively put, one can read $M \Downarrow V$ as stating that M is the program that we are running, and V is the result we get from it at the end.

The first part of our definition is that values evaluate to themselves:

$$\frac{}{V \Downarrow V}$$

As a consequence, we do not evaluate under λ-abstractions. Pragmatically, evaluating under lambda is closer to program optimization than to program execution. For instance, a compiler may inline function bodies, which is a form of beta reduction. In the operational semantics we only describe the program execution at runtime.

3.1 Call-by-name evaluation

In call-by-name, we evaluate an application $(\lambda x.M_1)\, M_2$ by evaluating M_1 after binding x to M_2. That evaluation strategy is formalized by the following rule:

$$\frac{M_1 \Downarrow (\lambda x.M_3) \qquad M_3[x \mapsto M_2] \Downarrow V}{(M_1\, M_2) \Downarrow V}$$

Note the substitution of M_2 *without* evaluating it first; that is what "by name" refers to.

Formally, such a rule says that if the statements above the horizontal line are true, then the one below the line is true as well. It is usually more intuitive to read the rule in a clockwise direction:

- We want to know what $M_1\,M_2$ evaluates to (on the bottom left);
- so we take the term apart and see what we get for M_1 (on the top left);
- then we go right and proceed with $M_3[x \mapsto M_2]$ (on the top right);
- we get a V and take that back down (to the bottom right).

The most widely known programming language with call-by-name semantics is Haskell.

Example: what happens when we try to evaluate the non-terminating term $(\lambda x.xx)\,(\lambda x.xx)$? It turns out there is *no* value V such that

$$(\lambda x.xx)\,(\lambda x.xx) \Downarrow V$$

Any attempt to derive such a V forces us into an infinite loop:

$$\frac{(\lambda x.xx) \Downarrow (\lambda x.xx) \qquad (\lambda x.xx)\,(\lambda x.xx) \Downarrow ?}{(\lambda x.xx)\,(\lambda x.xx) \Downarrow ?}$$

3.2 Call-by-value evaluation

In call-by-value, the parameter to a function is evaluated to a value V before the function is called. The vast majority of programming languages use call-by-value semantics, including Java.

We need to compute what to call, what to call it with, and the call itself. In call-by-value, the evaluation of an application $M_1 M_2$ consists of three steps:

1. evaluate the operator M_1
2. evaluate the operand (actual parameter) M_2
3. evaluate the body of M_1 after passing the value of M_2.

Consequently, the rule for evaluating an application is now:

$$\frac{M_1 \Downarrow (\lambda x.M_3) \qquad M_2 \Downarrow V_1 \qquad M_3[x \mapsto V_1] \Downarrow V_2}{(M_1\,M_2) \Downarrow V_2}$$

Note the substitution of the *value* of M_2; that is what "by value" refers to. The rules for big-step call-by-value evaluation are given in Figure 3.

Consider an application like this:

$$(M_1\,M_2)\,M_3$$

We have to evaluate $M_1\,M_2$ before we can apply the result to M_3. In old-timey imperative languages like C, that situation is rare. One tends to have things like `f(g(h()))`, but applications like `f(g)(h)` are uncommon. In Java, however, one often has things like

```
fooFactory.makeFoo(bar).setupFoo(qux).doSomeWorkFoo(wubble);
```

Exercise 3.1 (Easy) Find V such that

$$((\lambda f.\lambda x.f(fx))\,(\lambda y.y)\,5) \Downarrow V.$$

holds.

Beta reduction:

$$\frac{}{((\lambda x.M_1)\,M_2) \to_\beta (M_1[x \mapsto M_2])}$$

Compatible closure of $\to_\beta$:

$$\frac{M_1 \to_\beta M_2}{(M_1\,M_3) \to_\beta (M_2\;M_3)} \qquad \frac{M_1 \to_\beta M_2}{(M_3\,M_1) \to_\beta (M_3\;M_2)} \qquad \frac{M_1 \to_\beta M_2}{(\lambda x.M_1) \to_\beta (\lambda x.M_2)}$$

Reflexive-transitive closure $\to_\beta^*$ of $\to_\beta$:

$$\frac{}{M \to_\beta^* M} \qquad \frac{M_1 \to_\beta^* M_2 \qquad M_2 \to_\beta^* M_3}{M_1 \to_\beta^* M_3}$$

Figure 1: Lambda calculus: definition of reduction sequences in all contexts

$$\frac{}{V \Downarrow V} \qquad \frac{M_1 \Downarrow (\lambda x.M_3) \qquad M_3[x \mapsto M_2] \Downarrow V}{(M_1\,M_2) \Downarrow V}$$

Figure 2: Big-step evaluation for call-by-name

$$\frac{}{V \Downarrow V} \qquad \frac{M_1 \Downarrow (\lambda x.M_3) \qquad M_2 \Downarrow V_1 \qquad M_3[x \mapsto V_1] \Downarrow V_2}{(M_1\,M_2) \Downarrow V_2}$$

Figure 3: Big-step evaluation for call-by-value

Theorem 3.2 $M \Downarrow V$ implies $M \to_\beta{}^* V$.

The proof is by induction on the derivation of a judgement $M \Downarrow V$. There are two cases, one for values and one for applications.

We consider values first. Suppose we have derived

$$\frac{}{V \Downarrow V}$$

In that case, we have a trivial reduction in 0 steps: $V \to_\beta{}^0 V$ and therefore $V \to_\beta{}^* V$ as required.

Now consider an application $M_1\, M_2$ and suppose we have derived

$$\frac{M_1 \Downarrow (\lambda x.M_3) \qquad M_2 \Downarrow V_1 \qquad M_3[x \mapsto V_1] \Downarrow V_2}{(M_1\, M_2) \Downarrow V_2}$$

We apply the induction hypothesis to all three premises, which allows us to replace $\Downarrow$ with $\to_\beta^*$, as follows:

$$M_1 \to_\beta{}^* (\lambda x.M_3) \qquad M_2 \to_\beta{}^* V_1 \qquad M_3[x \mapsto V_1] \to_\beta{}^* V_2$$

As β-reduction is compatible, we also have:

$$M_1\, M_2 \to_\beta{}^* (\lambda x.M_3)\, M_2 \qquad (\lambda x.M_3)\, M_2 \to_\beta{}^* (\lambda x.M_3)\, V_1$$

We combine these reduction sequences with one more β reduction:

$$M_1\, M_2 \to_\beta{}^* (\lambda x.M_3) V_1 \qquad \to_\beta{}^1 \qquad M_3[x \mapsto V_1] \to_\beta{}^* V_2$$

This gives us the reduction sequence

$$(M_1\, M_2) \to_\beta{}^* V_2$$

as required.

Exercise 3.3 (Hard) Given a term M, can there be two different V_1 and V_2 such that

$$M \Downarrow V_1 \text{ and } M \Downarrow V_2$$

holds?

3.3 Adding assignment to the language

We now extend the language with mutable state. There are different ways of doing this; the one used here is based on the Definition of Standard ML[1] We add new constructs for creating references, reading their value, and updating them.

Adding references to lambda calculus radically changes what kind of programs we can write. In particular, we can do a form of object-oriented programming with classes. The canonical example is a "factory" that produces new objects, where an object is simply a pair consisting of a `set` and `get` method.

In OCaml we can write an object factory like this:

[1] `http://www.lfcs.inf.ed.ac.uk/reports/88/ECS-LFCS-88-62/ECS-LFCS-88-62.pdf`, page 49

$$\frac{n \notin \mathrm{dom}(s)}{s \vdash \mathtt{ref}\, V \Downarrow n, s[n \mapsto V]} \qquad \frac{}{s \vdash\, !\, n \Downarrow \mathtt{lookup}\, n\, \mathtt{in}\, s, s}$$

$$\frac{}{s \vdash (n := V) \Downarrow V, s[n \mapsto V]} \qquad \frac{}{s \vdash V \Downarrow V, s}$$

$$\frac{s_1 \vdash M_1 \Downarrow (\lambda x.M_3), s_2 \qquad s_2 \vdash M_2 \Downarrow V_1, s_3 \qquad s_3 \vdash M_3[x \mapsto V_1] \Downarrow V_2, s_4}{s_1 \vdash (M_1\, M_2) \Downarrow V_2, s_4}$$

$$\frac{s_1 \vdash M_1 \Downarrow n_1, s_2 \qquad s_2 \vdash M_2 \Downarrow n_2, s_3}{s_1 \vdash (M_1 + M_2) \Downarrow (n_1 + n_2), s_3}$$

Figure 4: Big-step evaluation for call-by-value with assignment

```
let factory x =
    let p = ref x in
        let get () = !p in
        let set y = p := y in
(get, set);;
```

In type systems, we need a context Γ in typing judgements $\Gamma \vdash M : \tau$. In an operational semantics, we can similarly add more structure, and the notation $\vdash$ is often borrowed for it.

For example, in a language with assignment, we need to say what the state is before and after evaluating a term. Our judgements then have the form

$$s_1 \vdash M \Downarrow V, s_2$$

The states s_1 and s_2 are finite partial functions from integers n to values V. A special case is the empty partial function $\emptyset$, which is undefined for all n. It models a state in which no addresses have been allocated.

We add three additional constructs to our lambda calculus for state manipulation: dereferencing, new allocation, and assignment to references. (We also add addition, $+$, which is not strictly necessary in this context, but convenient in writing examples.)

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & !\, V \\ & | & \mathtt{ref}\, V \\ & | & V_1 := V_2 \\ & | & M_1 + M_2 \end{array}$$

For a value V, the term $\mathtt{ref}\, V$ allocates a fresh storage cell n and then stores V into it:

$$\frac{n \notin \mathrm{dom}(s)}{s \vdash \mathtt{ref}\, V \Downarrow n, s[n \mapsto V]}$$

Note that $\Downarrow$ is not deterministic due to this rule, as there is a choice of locations to allocate.

Given a storage cell n, the term $\texttt{!}\, n$ reads its value from the store:

$$\frac{}{s \vdash \texttt{!}\, n \Downarrow \texttt{lookup}\, n\, \texttt{in}\, s, s}$$

Given a cell n and a value V, we have the usual assignment operation $n \texttt{:=} V$.

$$\frac{}{s \vdash (n \texttt{:=} V) \Downarrow V, s[n \mapsto V]}$$

Note that we need to fix an evaluation order, such as call-by-value with left-to-right evaluation, for assignments to be useful. Otherwise, the state changes could happen in an unspecified order.

Here is how we string the state changes together for call-by-value, left-to-right evaluation order:

$$\frac{s_1 \vdash M_1 \Downarrow (\lambda x.M_3), s_2 \qquad s_2 \vdash M_2 \Downarrow V_1, s_3 \qquad s_3 \vdash M_3[x \mapsto V_1] \Downarrow V_2, s_4}{s_1 \vdash (M_1\, M_2) \Downarrow V_2, s_4}$$

Note how the states s_j are pushed around in a clockwise direction, while being changed from s_j to s_{j+1}.

For values V, there is no state change:

$$\frac{}{s \vdash V \Downarrow V, s}$$

The rules for big-step call-by-value evaluation with assignment are collected in Figure 4.

As it stands, the language is untyped. We could try to dereference an integer, as in $\texttt{!}\, 42$, or we could take a reference and treat it as an integer, as in $42 + \texttt{ref}\, (\lambda x.x)$. In ML, such ill-formed programs are ruled out by the type system. We have not defined types yet, but the intention is that references are kept separate from integers. We can only dereference or assign to something if it has the appropriate reference type. The only way to make references is by allocating them with $\texttt{ref}\, V$.

Note that the evaluation gets stuck given terms like

$$(\lambda x.x \texttt{:=} 1)\, (\lambda y.y)$$

In a real implementation, this situation could lead to an exception being raised, or a function being corrupted. In ML, the type system prevents ill-typed code from being compiled.

We define syntactic sugar for local definitions and sequencing:

$$\begin{aligned} \texttt{let}\, x = M_1\, \texttt{in}\, M_2 &= (\lambda x.M_2)\, M_1 \\ (M_1; M_2) &= (\lambda d.M_2)\, M_1 \end{aligned}$$

Here d is a "dummy" variable, i.e., one that is not free in M_2.

Exercise 3.4 Give big-step rules with state for local definition and sequencing that express their evaluation directly, without translating them to λ. Hint: you

only need one rule for each of these constructs. In both cases, you need to start by evaluating M_1. Thus

$$s_1 \vdash M_1 \Downarrow V, s_2$$

is the first thing above the horizontal line in the rules.

Exercise 3.5 Could we also define local definition via substitution:

$$\texttt{let}\ x = M_1\ \texttt{in}\ M_2 \quad = \quad M_2[x \mapsto M_1]$$

Does it make a difference?

Exercise 3.6 Let L be the following term:

$$L = \lambda x.\texttt{let}\ p = (\texttt{ref}\ x)\ \texttt{in}\ \lambda m.(p\ \texttt{:=}\ \texttt{!}\ p + m)$$

Evaluate the following term:

$$\texttt{let}\ f = (L\ 1)\ \texttt{in}\ (f\ 2; f\ 2)$$

That is, find a value V and state s_2 such that

$$\emptyset \vdash \texttt{let}\ f = (L\ 1)\ \texttt{in}\ (f\ 2; f\ 2) \Downarrow V, s_2$$

In fact, $V = 5$, but you need to prove it.

The derivation tree is quite large, and you need to break it down into manageable subtrees. You may find it useful to use the following abbreviations. Let us write

$$M_a = \lambda m.a\ \texttt{:=}\ \texttt{!}\ a + m$$

for the function you get as the value for f, where a is the address of the local state. For any constant n, we write s_n for the state

$$s_n = \{a \mapsto n\}$$

It is useful to note that we have the following evaluation for any integer n and m:

$$s_n \vdash (M_a\ m) \Downarrow (n + m), s_{n+m}$$

4 The CEK machine

The best known abstract machine is the SECD machine, invented by Peter Landin in 1964. The CEK machine is a simpler and more recent machine (from 1985) that is explicitly built on the idea of continuation. The name CEK is due to the fact that the machine has three components:

- C stands for control or code. Intuitively, it is the expression that the machine is currently trying to evaluate.
- E stand for environment. It gives bindings for the free variables in C.
- K stands for continuation (given that the letter C is already used up). It tells the machine what to do when it is finished the with the current C.

The CEK machine evaluate λ-calculus expressions, which gives one the core of a programming language. One can then add additional constructs by extending the machine.

The CEK machine as presented here machine evaluate lambda terms under a call-by-value strategy. Abstract machines also exist for call-by-name, Krivine's abstract machine being perhaps the leading example.

The language that the basic CEK machine can interpret is λ calculus with constants n:

$$\begin{array}{rcl} M & ::= & x \\ & | & M_1 M_2 \\ & | & \lambda x.M \\ & | & n \end{array}$$

A term in the λ-calculus can have free variables. To evaluate the term, we need to have values for those free variables. The data structure that contains these variable bindings is called an environment. When a term is passed around as a value, it is necessary to pack it up with all its current variable bindings in a data structure called a closure. A closure is of the form

$$\texttt{clos}(\lambda x.M, E)$$

Here x is called the parameter, M the body, and E the environment of the closure.

A value W can be a constant n or a closure:

$$\begin{array}{rcl} W & ::= & n \\ & | & \texttt{clos}(\lambda x.M, E) \end{array}$$

An environment E is a list of the form

$$x_1 \mapsto W_1, \ldots, x_n \mapsto W_n$$

that associates the value W_j to variable x_j. We write $\emptyset$ for the special case when the list is empty. We can look up the value W_j for variable x_j in E, which we write as $E(x_j)$. An environent E can be updated with a new binding, giving x the value W, which we write as

$$E[x \mapsto W]$$

The machine needs to keep track of where in some bigger term it is currently working (what the continuation of the current term is). A frame is of the form

$$\begin{array}{rcl} F & ::= & (W \bigcirc) \\ & | & (\bigcirc\, M\, E) \end{array}$$

Intuitively, a frame is like an application in λ calculus with a hole $\bigcirc$ in the left or right position (the operator or the operand, respectively).

A continuation K is a stack of frames. If F is a frame and K a continuation, we write F, K for the continuation which we get by pushing F onto the top of K. The empty stack is written as $\blacksquare$.

$$\langle x \mid E \mid K \rangle \rightsquigarrow \langle \texttt{lookup}\, x\, \texttt{in}\, E) \mid E \mid K \rangle \quad (1)$$

$$\langle M_1\, M_2 \mid E \mid K \rangle \rightsquigarrow \langle M_1 \mid E \mid (\bigcirc\, M_2\, E), K) \rangle \quad (2)$$

$$\langle \lambda x.M \mid E \mid K \rangle \rightsquigarrow \langle \texttt{clos}(\lambda x.M, E) \mid E \mid K \rangle \quad (3)$$

$$\langle W \mid E_1 \mid (\bigcirc\, M\, E_2), K \rangle \rightsquigarrow \langle M \mid E_2 \mid (W\, \bigcirc), K \rangle \quad (4)$$

$$\langle W \mid E_1 \mid (\texttt{clos}(\lambda x.M, E_2)\, \bigcirc), K \rangle \rightsquigarrow \langle M \mid E_2[x \mapsto W] \mid K \rangle \quad (5)$$

Figure 5: CEK machine transition steps

A frame $(W\, \bigcirc)$ on the stack means that a result should be plugged into the hole position on the right, so that the value W is then applied to it. Almost symmetrically, a frame $(\bigcirc\, M\, E)$ means that the result should be plugged in on the left so as to be applied to M. However M should be evaluated first, and we need E in the frame for evaluating M relative to E.

A configuration $\langle M \mid E \mid K \rangle$ of the CEK machine consists of a λ-term M, an environment E and a continuation K.

The transitions of the CEK machine are given in Figure 5.

It is important to understand why each transition step must be the way it is.

The first three rules make a case distinction based on the current control C:

1. If the current code is a variable x, we must look it up in the environment E.

2. If the current code is an application $M_1 M_2$, then we proceed to evaluate the M_1 in the operator position. M_2 is pushed onto the continuation stack. Note that M_2 may contain free variables. For looking them up later, we push the current environment E along with M_2.

3. If the current code is a lambda abstraction, we package it up with the current environment to form a closure. That closure is the value of the expression.

If the current expression is a value W, we have evaluated it as far as it goes. In that case, we pop off the top of the continuation stack K. We make a case distinction depending on whether the top frame tells us we are in the operator position (with the hole on the left) or the operand position (with the hole on the right).

4. If the hole in the top frame is on the left, we pop of the term M and its associated environment E, and proceed to evaluate M relative to E. We also push on a new frame that tells us we still have to apply W to the result we hope to get from the evalutation of M.

5. If the hole in the top frame is on the right, we pop off the frame, which should contain a closure. We apply the closure by evaluating its body in the environment extended with the value W for the argument of the closure. If there is no closure in the frame, then the machine gets stuck and there is no transition step. That situation arises if we get a run-time

type error from ill-typed code such as applying a constant to something, as in $7\,(\lambda x.x)$.

Here is how the CEK machines evaluates the application

$$(\lambda x.\lambda y.x)\,1\,2$$

to the value 1.

$$\begin{array}{rl}
 & \langle\,(\lambda x.\lambda y.x)\,1\,2 \;\mid \emptyset \mid \blacksquare\,\rangle \\
\rightsquigarrow & \langle\,(\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,(\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\,1\,\emptyset), (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,\mathtt{clos}((\lambda x.\lambda y.x), \emptyset) \;\mid \emptyset \mid (\bigcirc\,1\,\emptyset), (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,1 \;\mid \emptyset \mid (\mathtt{clos}((\lambda x.\lambda y.x), \emptyset)\,\bigcirc), (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,\lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,\mathtt{clos}((\lambda y.x), x \mapsto 1) \;\mid x \mapsto 1 \mid (\bigcirc\,2\,\emptyset)\,\rangle \\
\rightsquigarrow & \langle\,2 \;\mid \emptyset \mid (\mathtt{clos}(\lambda y.x, x \mapsto 1)\,\bigcirc)\,\rangle \\
\rightsquigarrow & \langle\,x \;\mid x \mapsto 1, y \mapsto 2 \mid \blacksquare\,\rangle \\
\rightsquigarrow & \langle\,1 \;\mid x \mapsto 1, y \mapsto 2 \mid \blacksquare\,\rangle
\end{array}$$

Exercise 4.1 (Easy) Explain what sequence of steps the CEK machine performs starting from the following configuration:

$$\langle\,(\lambda x.xx)\,(\lambda x.xx) \;\mid \emptyset \mid \blacksquare\,\rangle$$

In particular, does one get a W at the end?

It is instructive to compare the big-step semantics in Figure 3 and the CEK machine in Figure 5. In the machine semantics, context information is made explicit in the machine state, whereas in the big-step semantics it is kept in the derivation tree rather than the terms themselves.

We say that a term M evaluates to a value W if there is a sequence of steps:

$$\langle\,M \;\mid \emptyset \mid \blacksquare\,\rangle \rightsquigarrow \cdots \rightsquigarrow \langle\,W \;\mid E \mid \blacksquare\,\rangle$$

We write $M \Downarrow_{\mathbf{CEK}} W$ in that case.

Theorem 4.2 If $M \Downarrow_{\mathbf{CEK}} n$ for some integer n, then $M \Downarrow n$ under call-by-value big-step evaluation.

Exercise 4.3 (Hard) Suppose we do not build closures at all, and we just pass around terms like $\lambda x.M$ instead. How would that affect the semantics of the language? (Hint: name capture).

Closures are fundamental for functional languages like Haskell, ML, Scheme, or Common Lisp. In Java, anonymous inner classes were introduced to build a kind of closure. In C#, delegates perform a similar function. The dialect of Lisp found in the Emacs editor is one of the few functional languages that does *not* construct closures when evaluating a lambda abstraction.

5 Adding control operators to the CEK machine

We add a simple form of non-local control structure to the CEK machine. It is an idealization of exceptions or the `setjmp()`/`longjmp()` construct in C.

We extend the programming language with two control operators:

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & \texttt{go}\, M \\ & | & \texttt{here}\, M \end{array}$$

Intuitively, `go` jumps to the nearest enclosing `here`.

To give meaning to the control operators, we add a new kind of frame. It just marks a position on the stack:

$$F ::= \ldots \mid \blacktriangleright\!\blacktriangleright$$

The CEK+`go` machine with control has these additional transition steps:

$$\langle\, \texttt{here}\, M \mid E \mid K \,\rangle \rightsquigarrow \langle\, M \mid E \mid \blacktriangleright\!\blacktriangleright, K \,\rangle \tag{6}$$

$$\langle\, \texttt{go}\, M \mid E \mid K_1, \blacktriangleright\!\blacktriangleright, K_2 \,\rangle \rightsquigarrow \langle\, M \mid E \mid K_2 \,\rangle \tag{7}$$

$$\langle\, W \mid E \mid \blacktriangleright\!\blacktriangleright, K \,\rangle \rightsquigarrow \langle\, W \mid E \mid K \,\rangle \tag{8}$$

Here K_1 does not contain $\blacktriangleright\!\blacktriangleright$, that is,

$$K_1 \neq K_3, \blacktriangleright\!\blacktriangleright, K_4$$

for any K_3 and K_4.

The idea behind the rules is as follows:

6. A term `here` M is evaluated by pushing the marker onto the stack and continuing with evaluating M.
7. A term `go` M is evaluated by erasing the stack up to the nearest marker, and then proceeding with evaluating M.
8. When a value W is produced by the evaluation, any marker on the top of the stack is popped off and ignored.

The following code shows that exceptions in OCaml also use the dynamically enclosing handler:

```
exception E;;

let f =
    try
        fun x -> raise E
    with E -> fun y -> "static"
in
    try
        f 0
    with E -> "dynamic"
;;
```

$$\langle x \mid E \mid K \rangle \rightsquigarrow \langle \texttt{lookup}\, x\, \texttt{in}\, E) \mid E \mid K \rangle \quad (1)$$
$$\langle M_1\, M_2 \mid E \mid K \rangle \rightsquigarrow \langle M_1 \mid E \mid (\bigcirc\, M_2\, E), K) \rangle \quad (2)$$
$$\langle \lambda x.M \mid E \mid K \rangle \rightsquigarrow \langle \texttt{clos}(\lambda x.M, E) \mid E \mid K \rangle \quad (3)$$
$$\langle W \mid E_1 \mid (\bigcirc\, M\, E_2), K \rangle \rightsquigarrow \langle M \mid E_2 \mid (W\, \bigcirc), K \rangle \quad (4)$$
$$\langle W \mid E_1 \mid (\texttt{clos}(\lambda x.M, E_2)\, \bigcirc), K \rangle \rightsquigarrow \langle M \mid E_2[x \mapsto W] \mid K \rangle \quad (5)$$
$$\langle\, \texttt{here}\, M \mid E \mid K \rangle \rightsquigarrow \langle M \mid E \mid \blacktriangleright\!\blacktriangleright, K \rangle \quad (6)$$
$$\langle\, \texttt{go}\, M \mid E \mid K_1, \blacktriangleright\!\blacktriangleright, K_2 \rangle \rightsquigarrow \langle M \mid E \mid K_2 \rangle \quad (7)$$
$$\langle W \mid E \mid \blacktriangleright\!\blacktriangleright, K \rangle \rightsquigarrow \langle W \mid E \mid K \rangle \quad (8)$$

Figure 6: CEK+go machine with control operators here and go

Exercise 5.1 (Straightforward calculation) Evaluate the following term in the CEK+go machine

$$(\lambda f.\, \texttt{here}\, ((\lambda x.1)\, (f\, 2)))\, (\texttt{here}\, (\lambda y.\, \texttt{go}\, y))$$

You should pay particular attention to which of the two occurrences of here is jumped to. The one on the left is *dynamically enclosing* (when the go is evaluated), whereas the one on the right is *statically enclosing* (where the go is defined).

Exercise 5.2 Write a Java program that illustrates that Java exceptions jump to the dynamically enclosing rather than the statically enclosing exception handler. You may use an anonymous inner class to get a statically enclosing exception handler around a method.

There are other, more powerful, control operators that do not merely erase parts of the stack, but copy it and pass it around as a piece of data, to be reinstated later.

6 Implementing an abstract machine

The CEK machine has many connections to compiler construction. In particular, the K component is closely related to the call stack. Programming language runtime systems also have to maintain data structures analogous to the environment E. The concept of closure was an important innovation in the SECD machine. Closures are fundamental for functional programming languages, and they are gradually spreading to other languages.

- The code component C can be represented as an abstract syntax tree of lambda calculus. Alternatively, it could be represented as compiled code.
- The environment component E can be represented as a linked list of pairs. We need to allocate new list nodes when our environment grows.
- The continuation component K can be represented as a stack of frames. Here a stack is an array with a stack pointer. The marker for the control operators is implemented as a pointer into the stack.

- Closures are represented as a pair of a pointer to code and and an environment pointer:

```
struct funval {
    struct exp *code;
    struct env *env;
};
```

See

`http://www.cs.bham.ac.uk/~hxt/2010/02552/go-here-interpreter.c`

for a toy implementation of a machine with `here` and `go`.

Carrying around the environment may seem like a lot of hassle. But on the other hand, we never have to rename bound variables, as one may have to do in the formal inductive definition of substitution.

In a real run-time system, we do not keep the names of bound variables at all. Instead, they are replaced by integers as placeholders. In that sense, the implementations resemble the use of deBruijn indices in the lambda calculus. For instance, in Java bytecode, parameters are numbered 0,1,2,.... Similarly, in C parameters work like array indices into the current stack frame.

Furthermore, the program is typically compiled before being run. For the control component of the machine, that means that the control does not really consist of a term that is decomposed and passed around. Instead, control would be represented by an instruction pointer.

7 Contextual equivalence

An operational semantics tells us what a program means in terms of what it does. We can then ask when two programs behave the same, so that they are interchangeable.

We will use integers n as the observable outcomes of program behaviour. In a big-step semantics, the semantics tells us directly what the outcome of a program M is via the relation $M \Downarrow n$.

For the CEK machine, we write $M \Downarrow n$ if

$$\langle M \mid \emptyset \mid \blacksquare \rangle \rightsquigarrow \cdots \rightsquigarrow \langle n \mid E \mid \blacksquare \rangle$$

Given $M \Downarrow n$, we can for example reason that

$$((\lambda x.\lambda y.x)\,1)\,2 \text{ and } ((\lambda y.\lambda x.x)\,2)\,1$$

behave the same.

Formally, a context C is a "term with a hole". For the lambda-calculus, the syntax of contexts is as follows:

$$\begin{array}{rcl} C & ::= & \bigcirc \\ & \mid & M\,C \\ & \mid & C\,M \\ & \mid & \lambda x.C \end{array}$$

If we have more constructs in the language, the grammar for contexts has to be extended accorsingly.

We write $C[M]$ for the term that we get by plugging M into the hole position $\bigcirc$ in C.

For a lambda-term M, a context C is called a *closing* context if $C[M]$ is a closed lambda term (i.e., it has not free variables).

Suppose we have an operational semantics given by $\Downarrow$. We say that M_1 and M_2 are *contextually equivalent*, written as $M_1 \simeq M_2$, if for all closing contexts C and constants n it is the case that

$$C[M_1] \Downarrow n \text{ if and only if } C[M_2] \Downarrow n$$

Intuitively, if two programs are contextually equivalent, they cannot be distinguished by any "client" that uses them. As a consequence, a compiler can replace M_1 by M_2 if it is more efficient. Similarly a refactoring tool rewrites code to something that ought to be contextually equivalent.

For any programming language, contextual equivalence gives us a rough-and-ready notion of what it means for two programs to behave the same. We only need to adapt the definition of context to the syntax of the language. Proving such contextual equivalences is quite hard, and various mathematical techniques have been developed.

Exercise 7.1 (Easy) Prove that contextual equivalence is an equivalence relation.

In the lambda calculus, we can ask whether beta reduction is a contextual equivalence:

$$\text{If } M_1 \to_\beta M_2 \text{ then } M_1 \simeq M_2$$

The equivalence may seen obvious, but proving it takes some care, and it may be necessary to restrict the beta reduction to the case where the argument is a value:

$$(\lambda x.M)\, V \;\to_{\beta V}\; M[x \mapsto V]$$

For example, we cannot perform β-reduction for non-values, as in

$$(\lambda x.M)\ (\texttt{go}\, 5)$$
$$(\lambda x.M)\ (\texttt{ref}\, 5)$$

Contextual equivalence changes due to exceptions. In lambda calculus, we have the following contextual equivalence:

$$(\lambda x.\Omega)\, M \simeq \Omega$$

where $\Omega = (\lambda x.xx)\,(\lambda x.xx)$. In a language with exceptions, the two sides can be distinguished.

Exercise 7.2 (Hard) Give a term M and a context C to show that exceptions break the equivalence.

8 Types and effects

Once we have defined an operational semantics, we can ask various questions about our programming language based on the semantics.

A basic sanity check for an operational semantics is that the evaluation preserves types. The result we would always expect is the following: If $\Gamma \vdash M : \tau$ and $M \Downarrow V$, then $\Gamma \vdash V : \tau$.

Technically, the rules for type systems resemble big-step semantics in that they generate proof trees. Compare the big-step evaluation rule with the typing rule for application:

$$\frac{\Gamma \vdash M_1 : (A \to B) \qquad \Gamma \vdash M_2 : A}{\Gamma \vdash (M_1\, M_2) : B}$$

Again we can read the rule clockwise: to type the application $(M_1\, M_2)$, we need to type the function M_1 and then the argument M_2.

To track exceptions, we can add effects to the type system:

$$\Gamma \vdash M : A \,!\, e$$

where e is the set of exceptions that the execution of M may produce. Note that Java has such a type-and-effect system, in the form of `throws` declarations in method types. For example,

```
public final boolean readBoolean() throws IOException;
```

In this case, `boolean` is the type and `IOException` is the effect. In Java, the burden of declaring the possible exceptions is placed on the programmer. There is some research that infers the possible exceptions automatically, in a form of effect inference analogous to type inference.

To prove correctness of an effect system for exceptions, we could relate it to runs of the CEK machine with exceptions.

9 A machine with a heap and pointers

We can also define abstract machines for imperative languages close to C or even assembly language. A big problem in current research is to reason about pointers. We define an abstract machine that can manipulate pointers, and that is as simple as possible otherwise.

The configurations of the machine are of the form

$$\langle c \mid h \rangle$$

where c is a sequence of commands and h is a heap.

A heap is a partial functions from integers to integers. Intuitively $h(n) = m$ means that the heap at address n contains the integer m. In particular, the contents of a heap cell may itself serve as the address of some other cell, so that we can build complex pointer structures in the heap.

Expressions can be integer constants n, the addition of two expressions, or a pointer dereferencing:

$$\begin{array}{rcl} e & ::= & n \\ & \mid & e_1 + e_2 \\ & \mid & !\, e \end{array}$$

$$
\begin{array}{rcl}
\langle e_1 \texttt{ := } e_2 \texttt{ ; } c \mid h\rangle & \rightsquigarrow & \langle c \mid h[n_1 \mapsto n_2]\rangle \\
\langle \texttt{malloc}(e_1, e_2) \texttt{ ; } c \mid h\rangle & \rightsquigarrow & \langle c \mid h[n_1 \mapsto 0, \ldots, (n_1 + n_2 - 1) \mapsto 0]\rangle \\
 & & \text{where } \{n_1, \ldots, n_1 + n_2 - 1\} \cap \mathrm{dom}(h) = \emptyset \\
\langle \texttt{while } e_1 \texttt{ do } c_1 \texttt{ ; } c_2 \mid h\rangle & \rightsquigarrow & \langle c_1 \texttt{ ; while } e_1 \texttt{ do } c_1 \texttt{ ; } c_2 \mid h\rangle \text{ if } n_1 \neq 0 \\
\langle \texttt{while } e_1 \texttt{ do } c_1 \texttt{ ; } c_2 \mid h\rangle & \rightsquigarrow & \langle c_2 \mid h\rangle \text{ if } n_1 = 0
\end{array}
$$

Figure 7: Transition steps for the heap machine

We write $h \vdash e \Downarrow n$ when n is the value of the expression e relative to the heap h. There are various ways of giving a precise definition of $h \vdash e \Downarrow n$. Here is one in big-step style:

$$
\frac{h \vdash e \Downarrow m \qquad h(m) = n}{h \vdash \texttt{!}\, e \Downarrow n} \qquad \frac{}{h \vdash n \Downarrow n}
$$

$$
\frac{h \vdash e_1 \Downarrow n_1 \qquad h \vdash e_2 \Downarrow n_2 \qquad n = n_1 + n_2}{h \vdash (e_1 + e_2) \Downarrow n}
$$

Commands can be assignments, memory allocations, or while loops. We define c to be a sequence of such commands, ended by `return`.

$$
\begin{array}{rcl}
c & ::= & \texttt{return} \\
 & \mid & e_1 \texttt{ := } e_2 \texttt{ ; } c \\
 & \mid & \texttt{malloc}(e_1, e_2) \texttt{ ; } c \\
 & \mid & \texttt{while } e \texttt{ do } c_1 \texttt{ ; } c_2
\end{array}
$$

Let $h \vdash e_1 \Downarrow n_1$ and $h \vdash e_2 \Downarrow n_2$. The transition steps of the machine are given in Figure 7.

Note that in a command sequence

$$c_1 \texttt{ ; } \ldots \texttt{ ; } c_n \texttt{ ; return}$$

the c_1 is analogous to the first component of the CEK machine, as it is what the machine is currently working on. The rest of the sequence, namely

$$c_2 \texttt{ ; } \ldots \texttt{ ; } c_n \texttt{ ; return}$$

represents what comes next, like the continuation (last component K) in the CEK machine, and the final `return` is like ■ in the CEK machine.

We are interested in the effect that a command sequence c has on the heap left after the evaluation steps:

$$\langle c \mid h_1\rangle \rightsquigarrow \ldots \rightsquigarrow \langle \texttt{return} \mid h_n\rangle$$

Exercise 9.1 (Easy) What does the heap look like after running the following code?

$$\texttt{malloc}(42, 2) \texttt{ ; ! } 42 + 1 \texttt{ := ! } 42 \texttt{ ; return}$$

We could define a type system for our language with pointers. For instance, one would expect something like

$$\frac{e : \texttt{pointer to } A}{\texttt{!}\, e : A}$$

The type system of C works like that (apart from its funny inside-out notation).

For languages with mutable state, Hoare logics have been very successful. In particular, Separation Logic allows us to describe the shape of the heap before and after execution.

An abstract machine for machine language (x86, ARM, SPARC or other architectures) would be more complex, but it would need some of the same features as the machine above, such as dereferencing pointers into the heap.

We could also add a heap to the CEK machine. That would give us a fairly powerful machine, which raises all kinds of research questions. The combination of lambda calculus functions and pointers is sufficient for a form of object-oriented programming, where the lambda abstractions give us methods and the pointers the private state of an object.

10 Concurrency

Operational semantics can also be extended to deal with concurrency. A common approach is to use interleaving. This is a reasonably good description of, say, Java threads on a single-processor machine. Interleaving is less apt for concurrent systems in which actions happen at the same time, as in a multi-core CPU or a GPU (graphics processor with many cores).

Concurrent interaction can be via message passing or a shared state.

10.1 Process calculi

We will not go into any detail about process calculi here, just enough to show how ideas from the lambda calculus can be extended in a new dimension, that of concurrent communication.

In process calculi like CCS and the pi calculus, we have a kind of generalized beta reduction in the form of message passing.

Recall beta reduction:

$$((\lambda x.M_1)\, M_2) \to_\beta (M_1[x \mapsto M_2])$$

We can read this reduction as a kind of communication, in which the λ-abstraction receives M_2 and binds it to x in its body M_1.

In a process calculus, such communication can occur repeatedly and along named channels. We communicate a value y along a channel a, so that it becomes the value of variable x in the receiving process:

$$(a(x).P) \parallel (\overline{a}(y).Q) \rightsquigarrow (P[x \mapsto y]) \parallel Q$$

Here y is sent along the channel a and substituted for x in the remainder of the process that received it. This form of communication turns out to be particularly powerful if the communicated value is itself a communication channel. The idea of transmitting channel names is the basis of the pi calculus.

Transition steps by different processes are interleaved:

$$\frac{P_1 \rightsquigarrow P_2}{(P_1 \parallel Q) \rightsquigarrow (P_2 \parallel Q)} \qquad \frac{P_1 \rightsquigarrow P_2}{(Q \parallel P_1) \rightsquigarrow (Q \parallel P_2)}$$

Here is an example of a process communication in the pi calculus:

$$\begin{aligned} & (\overline{f}(xk).k(w).P) \parallel (f(yh).\overline{h}(y).Q) \\ \rightsquigarrow \quad & (k(w).P) \parallel (\overline{k}(z).Q) \\ \rightsquigarrow \quad & P[w \mapsto z] \parallel Q \end{aligned}$$

The channel k is communicated along the channel f, enabling further communication along k.

Lambda calculus can be translated into the pi calculus. The basic idea is to add a return address to each function call. A lambda term $\lambda x.M$ is transformed into a process $f(xk).M$ listening on f for some x and k. For instance, the term $\lambda x.x$ is translated essentially as

$$f(xk).\overline{k}(x)$$

More technically, these return addresses function as continuations, like the stack in the CEK machine.

We can reason about contextual equivalence in the pi calculus much as in the lambda calculus. One application of such equivalences is to prove the correctness of security protocols: an attacker cannot observe the difference between two terms in the pi calculus, so that a secret is not leaked to the attacker.

10.2 Shared-state concurrency

In shared-state concurrency, we have a shared heap on which more than one program can perform reads and writes.

We can run two programs in parallel by interleaving the state changes they make on the shared heap:

$$\frac{\langle (c_1 \mid h_1) \rangle \rightsquigarrow \langle c_2 \mid h_2 \rangle}{\langle (c_1 \parallel c_3) \mid h_1 \rangle \rightsquigarrow \langle (c_2 \parallel c_3) \mid h_2 \rangle} \qquad \frac{\langle (c_1 \mid h_1) \rangle \rightsquigarrow \langle c_2 \mid h_2 \rangle}{\langle (c_3 \parallel c_1) \mid h_1 \rangle \rightsquigarrow \langle (c_3 \parallel c_2) \mid h_2 \rangle}$$

When two processes access the same memory and at least one of the access is a write, there is a *race condition*. Let c_1 and c_2 be defined as follows:

$$\begin{aligned} c_1 \quad &= \quad \texttt{!}\,3 \texttt{ := } \texttt{!}\,2 \texttt{ ; } \texttt{!}\,2 \texttt{ := } \texttt{!}\,3 + 10 \texttt{ ; return} \\ c_2 \quad &= \quad \texttt{!}\,4 \texttt{ := } \texttt{!}\,2 \texttt{ ; } \texttt{!}\,2 \texttt{ := } \texttt{!}\,4 + 10 \texttt{ ; return} \end{aligned}$$

In isolation, both of these program increment the heap cell `! 2` by 10.

There are several possibilities for the value of `! 2` in the heap h_2 that is obtained by interleaving the execution of c_1 and c_2:

$$\begin{aligned} & \langle (c_1 \parallel c_2) \mid h_1 \rangle \\ \rightsquigarrow \ldots \rightsquigarrow \quad & \langle (\texttt{return} \parallel \texttt{return}) \mid h_2 \rangle \end{aligned}$$

It is possible to reason about highly concurrent programs using program logics. For example, the Java library contains concurrent data structures that need to be verified.

Although operational semantics and abstract machines can be extended from a sequential to a concurrent setting, it is not longer enough to look at the meaning of programs in terms of a relation between an initial and a final machine states, as in a big-step semantics. Rather than relating an initial to a final heap, we also need to keep track of all the intermediate heaps that other processes could see, which is known as a *trace semantics*.

Some challenging problems in current research arise from programming multi-core processors and weak memory models.

11 Further reading

The idea of building objects not from classes but from lambda and assignment is elaborated in Abelson and Sussman's well-known textbook, often called "SICP" [AwJS96].

"The Definition of Standard ML" uses a big-step semantics for a whole programming language, Standard ML [MTHM97].

The SECD machine was introduced in Peter Landin's paper "The Mechanical Evaluation of Expressions" [Lan64]. Landin later added the J-operator to the SECD machine to provide a form of non-local jumping [Lan98, Thi98]. The `gohere` construct is from [Thi02]. Continuations were hinted at; for an introduction, see [Thi99]. The CEK machine evolved from the SECD machine via John Reynolds's "Definitional Interpreters" [Rey72] and Matthias Felleisen and Dan Friedman's work on continuations [FF86]. More recently, Olivier Danvy and his coauthors have investigated the CEK machine in relation to continuation passing in the lambda calculus; the version of the CEK machine presented here is inspired by theirs. Friedman, Wand and Haynes present interpreters analogous to the CEK machine in their textbook "Essentials of Programming Languages" [FWH92].

The big-step semantics and the abstract machine style are at opposite ends of a range of styles of operational semantics. There are others that are between these extremes, such as Structural Operational Semantics (SOS) and small-step, evaluation context semantics. In current research, a style of operational semantics with evaluation contexts represented as frame stacks, like those in the CEK machine, is often preferred, for example in Andy Pitts's work on observational equivalence. For process calculi, see Robin Milner's introduction to the pi calculus [Mil99]. For the Warren abstract machine, see Ait-Kaci's rational reconstruction [AK91].

Acknowledgements

Thanks to Maxim Strygin for texing up some solutions to the exercises.

References

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[AwJS96] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass., 1996.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.

[Lan65] Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965.

[Lan98] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [Lan65].

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, August 1972.

[Thi98] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[Thi99] Hayo Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, June 1999.

[Thi02] Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):141–160, 2002.

Appendix: solutions to selected exercises

Solution for Exercise 3.1

$$\frac{(\lambda f.\lambda x.f(fx))\,(\lambda x.x) \Downarrow \lambda y.(\lambda x.x)((\lambda x.x)y) \qquad \overline{5 \Downarrow 5} \qquad (\lambda x.x)((\lambda x.x)y)[y \mapsto 5] \Downarrow 5}{((\lambda f.\lambda x.f(fx))\,(\lambda x.x))\,5 \Downarrow 5}$$

where:

$$\dfrac{\dfrac{}{(\lambda f.\lambda x.f(fx)) \Downarrow (\lambda f.\lambda x.f(fx))} \quad \dfrac{}{\lambda x.x \Downarrow \lambda x.x} \quad \dfrac{}{(\lambda y.f(fy))[f \mapsto \lambda x.x] \Downarrow \lambda y.(\lambda x.x)((\lambda x.x)y)}}{(\lambda f.\lambda x.f(fx))\,(\lambda x.x) \Downarrow \lambda y.(\lambda x.x)((\lambda x.x)y)}$$

and:

$$\dfrac{\dfrac{}{(\lambda x.x) \Downarrow (\lambda x.x)} \quad \dfrac{\dfrac{}{(\lambda x.x) \Downarrow (\lambda x.x)} \quad \dfrac{}{5 \Downarrow 5} \quad \dfrac{}{x[x \mapsto 5] \Downarrow 5}}{(\lambda x.x)5 \Downarrow 5} \quad \dfrac{}{x[x \mapsto 5] \Downarrow 5}}{(\lambda x.x)((\lambda x.x)y)[y \mapsto 5] \Downarrow 5}$$

Solution for exercise 5.1

$\langle\, (\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)))\,(\mathtt{here}\,(\lambda y.\,\mathtt{go}\,y)) \;\mid \emptyset \mid \blacksquare \,\rangle$

(2) $\rightsquigarrow$ $\langle\, (\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2))) \;\mid \emptyset \mid (\bigcirc\,(\mathtt{here}\,(\lambda y.\,\mathtt{go}\,y))\,\emptyset) \,\rangle$

(3) $\rightsquigarrow$ $\langle\, \mathtt{clos}(\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)), \emptyset) \;\mid \emptyset \mid (\bigcirc\,(\mathtt{here}\,(\lambda y.\,\mathtt{go}\,y))\,\emptyset) \,\rangle$

(4) $\rightsquigarrow$ $\langle\, (\mathtt{here}\,(\lambda y.\,\mathtt{go}\,y)) \;\mid \emptyset \mid (\mathtt{clos}(\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)), \emptyset)\,\bigcirc) \,\rangle$

(6) $\rightsquigarrow$ $\langle\, (\lambda y.\,\mathtt{go}\,y) \;\mid \emptyset \mid \blacktriangleright\!\blacktriangleright, (\mathtt{clos}(\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)), \emptyset)\,\bigcirc) \,\rangle$

(3) $\rightsquigarrow$ $\langle\, \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \;\mid \emptyset \mid \blacktriangleright\!\blacktriangleright, (\mathtt{clos}(\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)), \emptyset)\,\bigcirc) \,\rangle$

(8) $\rightsquigarrow$ $\langle\, \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \;\mid \emptyset \mid (\mathtt{clos}(\lambda f.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)), \emptyset)\,\bigcirc) \,\rangle$

(5) $\rightsquigarrow$ $\langle\, \mathtt{here}\,((\lambda x.1)\,(f\,2)) \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid \blacksquare \,\rangle$

(6) $\rightsquigarrow$ $\langle\, ((\lambda x.1)\,(f\,2)) \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid \blacktriangleright\!\blacktriangleright \,\rangle$

(2) $\rightsquigarrow$ $\langle\, (\lambda x.1) \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid (\bigcirc\,(f2)\,f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset)), \blacktriangleright\!\blacktriangleright \,\rangle$

(3) $\rightsquigarrow$ $\langle\, \mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset)) \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid K \,\rangle$
where $K = (\bigcirc\,(f2)\,f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset)), \blacktriangleright\!\blacktriangleright$

(4) $\rightsquigarrow$ $\langle\, f2 \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid (\mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))\,\bigcirc), \blacktriangleright\!\blacktriangleright \,\rangle$

(2) $\rightsquigarrow$ $\langle\, f \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid K \,\rangle$
where $K = (\bigcirc\,2\,(f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))), (\mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))\,\bigcirc), \blacktriangleright\!\blacktriangleright$

(1) $\rightsquigarrow$ $\langle\, \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid K \,\rangle$
where $K = (\bigcirc\,2\,(f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))), (\mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))\,\bigcirc), \blacktriangleright\!\blacktriangleright$

(4) $\rightsquigarrow$ $\langle\, 2 \;\mid f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset) \mid K \,\rangle$
where $K = (\mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset)\,\bigcirc), (\mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))\,\bigcirc), \blacktriangleright\!\blacktriangleright$

(5) $\rightsquigarrow$ $\langle\, \mathtt{go}\,y \;\mid y \mapsto 2 \mid (\mathtt{clos}(\lambda x.1, f \mapsto \mathtt{clos}(\lambda y.\,\mathtt{go}\,y, \emptyset))\,\bigcirc), \blacktriangleright\!\blacktriangleright \,\rangle$

(7) $\rightsquigarrow$ $\langle\, y \;\mid y \mapsto 2 \mid \blacksquare \,\rangle$

(1) $\rightsquigarrow$ $\langle\, 2 \;\mid y \mapsto 2 \mid \blacksquare \,\rangle$