Abstract machines, control, and sequents

Pierre-Louis Curien (CNRS – Université Paris VII)

No Institute Given

Abstract. We describe simple call-by-value and call-by-name abstract machines, expressed with the help of Felleisen's evaluation contexts, for a toy functional language. Then we add a simple control operator and extend the abstract machines accordingly. We give some examples of their use. Then, restricting our attention to the sole core (typed) λ-calculus fragment augmented with the control operator, we give a logical status to the machinery. Evaluation contexts are typed "on the left", as they are directed towards their hole, or their input, in contrast to terms, whose type is that of their output. A machine state consists of a term and a context, and corresponds logically to a cut between a formula on the left (context) and a formula on the right (term). Evaluation, viewed logically, is cut-elimination: this is the essence of the so-called Curry-Howard isomorphism. Control operators correspond to classical reasoning principles, as was first observed by Griffin.

1 A simple call-by-value evaluator

Consider the following simple functional programming language, whose data types are (nonnegative) integers and lists.

$$\begin{aligned} M ::= {} & x \mid n \mid \mathtt{T} \mid \mathtt{F} \mid \mathit{nil} \mid ?l \mid \mathtt{h}(l) \mid \mathtt{t}(l) \mid M \mathit{\ op\ } M \\ & M \mapsto [\, M \,,\, M \,] \mid MM \mid \lambda x.M \mid Yf.M \ . \end{aligned}$$

Here *op* denotes collectively operations such as addition, multiplication, consing (notation $a \cdot l$), or equality test of two integers (notation $(m = n)$); *nil* is the empty list, $?l$ tests whether l is empty (i.e., $?\mathit{nil}$ evaluates to $\mathtt{T}$ and $?(a \cdot l)$ evaluates to $\mathtt{F}$), $\mathtt{h}(l)$ and $\mathtt{t}(l)$ allow us to retrieve the first element of a list and the rest of the list, respectively; $M \mapsto [\, N \,,\, P \,]$ passes control to N (resp. P) if M evaluates to $\mathtt{T}$ (resp. $\mathtt{F}$); MN is function application; $\lambda x.M$ is function abstraction; finally, $Yf.M$ denotes a recursive function definition. The more familiar construct ($\mathtt{let\ rec}\ fx = M\ \mathtt{in}\ N$) is defined from it as $(\lambda f.N)(Yf.(\lambda x.M))$.

Next, we specify an interpreter for the mini-language. The interpreter progressively transforms the whole program to be evaluated, and at each step maintains a pointer to a subprogram, in which the current work is done. Felleisen and Friedman [4] have formalized this using evaluation contexts, which are programs with a (single) hole that in our case are built recursively as follows:

$$\begin{aligned} E ::= {} & [\,] \mid E[[\,]M] \mid E[M[\,]] \\ & \mid E[[\,] \mathit{\ op\ } M] \mid E[M \mathit{\ op\ } [\,]] \mid E[\mathtt{h}([\,])] \mid E[\mathtt{t}([\,])] \mid E[?[\,]] \mid E[[\,] \mapsto [\, M \,,\, M \,]] \end{aligned}$$

The notation should be read as follows: E is a term with a hole, and, say, $E[[\,]M]$ is the context whose single occurrence of $[\,]$ has been replaced by $[\,]M$: thus, in $E[[\,]M]$, the external square brackets refer to the hole of E, while the internal ones refer to the hole of $E[[\,]M]$. For example, $[\,][[\,]M] = [\,]M$, and if this context is called E, then $E[[\,]N] = ([\,]N)M$. (The above syntax is in fact too liberal, see exercise 2.)

The abstract machine rewrites pairs (term,context) that we write $\langle\, N \mid E \,\rangle$ to stress the interaction (and the symmetry) of terms and contexts. We call such pairs $\langle\, N \mid E \,\rangle$ *states*, or *commands*. The initial command is of the form $\langle\, M \mid [\,] \,\rangle$. The rules (in call-by-value) are as follows:

$$
\begin{array}{lll}
\langle\, MN \mid E \,\rangle & \to \langle\, M \mid E[[\,]N] \,\rangle & \\
\langle\, \lambda x.P \mid E[[\,]N] \,\rangle & \to \langle\, N \mid E[(\lambda x.P)[\,]] \,\rangle & \\
\langle\, V \mid E[(\lambda x.P)[\,]] \,\rangle & \to \langle\, P[x \leftarrow V] \mid E \,\rangle & \\
 & & \\
\langle\, Yf.M \mid E \,\rangle & \to \langle\, M[f \leftarrow Yf.M] \mid E \,\rangle & \\
 & & \\
\langle\, M \mathbin{op} N \mid E \,\rangle & \to \langle\, M \mid E[[\,] \mathbin{op} N] \,\rangle & \\
\langle\, m \mid E[[\,] \mathbin{op} N] \,\rangle & \to \langle\, N \mid E[m \mathbin{op} [\,]] \,\rangle & \\
\langle\, n \mid E[m \mathbin{op} [\,]] \,\rangle & \to \langle\, m \mathbin{op} n \mid E \,\rangle & \text{(operation performed)} \\
 & & \\
\langle\, \star(M) \mid E \,\rangle & \to \langle\, M \mid E[\star([\,])] \,\rangle & (\star = ?, \mathtt{h}, \mathtt{t}) \\
\langle\, nil \mid E[?([\,])] \,\rangle & \to \langle\, \mathtt{T} \mid E \,\rangle & \\
\langle\, a \cdot l \mid E[?([\,])] \,\rangle & \to \langle\, \mathtt{F} \mid E \,\rangle & \\
\langle\, a \cdot l \mid E[\mathtt{h}([\,])] \,\rangle & \to \langle\, a \mid E \,\rangle & \\
\langle\, a \cdot l \mid E[\mathtt{t}([\,])] \,\rangle & \to \langle\, l \mid E \,\rangle & \\
\langle\, M \mapsto [\,N\,,\,P\,] \mid E \,\rangle & \to \langle\, M \mid E[[\,] \mapsto [\,N\,,\,P\,]] \,\rangle & \\
\langle\, \mathtt{T} \mid E[[\,] \mapsto [\,N\,,\,P\,]] \,\rangle & \to \langle\, N \mid E \,\rangle & \\
\langle\, \mathtt{F} \mid E[[\,] \mapsto [\,N\,,\,P\,]] \,\rangle & \to \langle\, P \mid E \,\rangle &
\end{array}
$$

The first rule amounts to moving the pointer to the left son: thus the evaluator is also left-to-right. The second rule expresses call-by-value: the argument N of the function $\lambda x.P$ must be evaluated before being passed to $\lambda x.P$. In the third rule, V denotes a value – that is, a function $\lambda x.P$, an integer n, or a list l which is either nil or $a \cdot l'$ where a is a value and l' is a value (later, we shall add more values) –, that can be passed, i.e., that can be substituted for the formal parameter x. The fourth rule allows us to unfold the definition of a recursive function. The last rules specify the (left-to-right) evaluation of the binary operations, and the precise meaning of the unary operations $?, \mathtt{h}$ and $\mathtt{t}$, as well as of $M \mapsto [\,N\,,\,P\,]$.

Notice that the above system of rules is deterministic, as at each step at most one rule may apply.

Exercise 1. Characterize the final states, i.e., the states that cannot be rewritten.

Exercise 2. Design a more restricted syntax for call-by-value evaluation contexts (hint: replace $E[M[\,]]$ by $E[V[\,]]$, etc...).

Remark 1. The call-by-name abstract machine is slightly simpler. The context formation rule $E[M[\,]]$ disappears, as well as the third rule above. The only rule which changes is the rule for $\lambda x.P$, which is now

$$\langle\, \lambda x.P \parallel E[[\,]N]\,\rangle \;\to\; \langle\, P[x \leftarrow N] \parallel E\,\rangle$$

i.e., N is passed unevaluated to the function $\lambda x.P$. All the other rules stay the same.

Remark 2. In call-by-name, the left-to-right order of evaluation given by the rule $\langle\, MN \parallel E\,\rangle \;\to\; \langle\, M \parallel E[[\,]N]\,\rangle$ is forced upon us: we should not attempt to evaluate N first. But in call-by-value, both M and N have to be evaluated, and the right-to-left order of evaluation becomes an equally valid strategy. In this variant, the first three rules are modified as follows:

$$\begin{array}{lll}
\langle\, MN \parallel E\,\rangle & \to & \langle\, N \parallel E[M[\,]]\,\rangle \\
\langle\, V \parallel E[M[\,]]\,\rangle & \to & \langle\, M \parallel E[[]V]\,\rangle \\
\langle\, \lambda x.P \parallel E[[\,]V]\,\rangle & \to & \langle\, P[x \leftarrow V] \parallel E\,\rangle
\end{array}$$

Below, we give a few examples of execution. We first consider a program that takes a natural number x as input and returns the product of all prime numbers not greater than x. One supposes given an operation $\pi?$ that tests its argument for primality, i.e., $\pi?(n)$ evaluates to $\mathtt{T}$ if n is prime, and to $\mathtt{F}$ if n is not prime. The program is a mere transcription of the specification of the problem:

$$\pi_\times \;=\; Yf.\lambda n.\,(n = 1) \mapsto [\,1\,,\,(\pi?(n) \mapsto [\,n \times f(n-1)\,,\,f(n-1)\,])\,]$$

Here is the execution of this program with input 4:

$$\begin{array}{lll}
\langle\, \pi_\times(4) \parallel [\,]\,\rangle & \to & \langle\, (4 = 1) \mapsto [\,1\,,\,(\pi?(4) \mapsto [\,4 \times \pi_\times(4-1)\,,\,\pi_\times(4-1)\,])\,] \parallel [\,]\,\rangle \\
& \to & \langle\, \pi?(4) \mapsto [\,4 \times \pi_\times(4-1)\,,\,\pi_\times(4-1)\,] \parallel [\,]\,\rangle \\
& \to & \langle\, \pi_\times(4-1) \parallel [\,]\,\rangle \\
& \to^\star & \langle\, \pi_\times(3) \parallel [\,]\,\rangle \\
& \to^\star & \langle\, 3 \times \pi_\times(3-1) \parallel [\,]\,\rangle \\
& \to^\star & \langle\, \pi_\times(2) \parallel 3 \times [\,]\,\rangle \\
& \to^\star & \langle\, \pi_\times(1) \parallel 3 \times (2 \times [\,])\,\rangle \\
& \to^\star & \langle\, 1 \parallel 3 \times (2 \times [\,])\,\rangle \\
& \to & \langle\, 2 \parallel 3 \times [\,]\,\rangle \\
& \to & \langle\, 6 \parallel [\,]\,\rangle
\end{array}$$

Our next example is the function that takes an integer n and a list l as arguments and returns l if n does not occur in l, or else the list of the elements of l found after the last occurrence of n in l. For example, when applied to 3 and $1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil}))))$, the function returns the list $4 \cdot \mathit{nil}$. The following program for this function makes use of an auxiliary list, that can be called an accumulator:

$$F \;=\; \lambda n.\lambda l.(Yf.\lambda l_1.\lambda l_2.\,?l_1 \mapsto [\,l_2\,,\,\mathtt{h}(l_1) = n \mapsto [\,f\,\mathtt{t}(l_1)\,\mathtt{t}(l_1)\,,\,f\,\mathtt{t}(l_1)\,l_2\,]\,])\,l\,l$$

We present the execution of F with inputs 3 and $1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil}))))$. We set:

$$\epsilon = Y f.\lambda l_1.\lambda l_2.?l_1 \mapsto [\, l_2 \,,\, \mathtt{h}(l_1) = 3 \mapsto [\, f\, \mathtt{t}(l_1)\, \mathtt{t}(l_1) \,,\, f\, \mathtt{t}(l_1)\, l_2 \,]\,]$$

We have:

$$\begin{array}{l}
\langle\, (F\,3)\,(1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil})))))\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,(1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil})))))\,(1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil})))))\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,(3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil}))))\,(1 \cdot (3 \cdot (2 \cdot (3 \cdot (4 \cdot \mathit{nil})))))\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,(2 \cdot (3 \cdot (4 \cdot \mathit{nil})))\,(2 \cdot (3 \cdot (4 \cdot \mathit{nil})))\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,(3 \cdot (4 \cdot \mathit{nil}))\,(2 \cdot (3 \cdot (4 \cdot \mathit{nil})))\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,(4 \cdot \mathit{nil})\,(4 \cdot \mathit{nil})\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, \epsilon\,\mathit{nil}\,(4 \cdot \mathit{nil})\, \mid [\,] \,\rangle \\
\quad \to^\star \langle\, 4 \cdot \mathit{nil}\, \mid [\,] \,\rangle
\end{array}$$

Note that the execution is tail-recursive: the evaluation context remains empty. This is good for efficiency, but, conceptually, handling the auxiliary list is somewhat "low level".

Remark 3. Similarly, our first example can be programmed in a tail recursive way, as

$$Y f.\lambda(n, c).\, n = 1 \mapsto [\, c(1) \,,\, f\,(n-1, \pi?(n) \mapsto [\, \lambda p.c(n \times p) \,,\, c\,])\,]$$

Here, c is an additional parameter, called the *continuation*, which is a function from natural numbers to natural numbers. This is the *continuation passing style* (CPS). We encourage the reader to run this new program on input $(4, \lambda x.x)$, and to check that the execution is indeed tail-recursive.

2 Control operators

We now add two primitive operations, in addition to those of the previous section:

$$M ::= \cdots \mid \kappa k.M \mid \star_E$$

The second construction allows us to consider, or *reflect* evaluation contexts as values (in addition to those considered above). It is then possible to bind a variable k to a (reflected) context, and thus to memorize and reuse contexts. This is what the first construction $\kappa k.M$ does. It exists in programming languages like SCHEME, where it is written as $(\mathtt{call/cc}\,(\mathtt{lambda}\,(k)\,M))$. We add two rules to the abstract machine (whether in call-by-value or in call-by-name):

$$\begin{array}{lll}
\langle\, \kappa k.M \mid E \,\rangle & \to & \langle\, M[k \leftarrow \star_E] \mid E \,\rangle \\
\langle\, \star_{E_1} \mid E_2[[\,]N] \,\rangle & \to & \langle\, N \mid E_1 \,\rangle
\end{array}$$

The second rule throws away the current evaluation context E_2 and replaces it with a context E_1 captured earlier using the first rule. Note that N is unevaluated (compare with the rule for λ which swaps function and argument).

We illustrate the new primitives through some examples. First, consider the function that takes as input a list of integers and returns the product of the elements of the list. A naïve program for this function is:

$$\Pi_1 \;=\; Yf.\lambda l.\; ?l \mapsto [\,1\,,\, \mathtt{h}(l) \times f(\mathtt{t}(l))\,]$$

The execution of this program applied to the list [2,4,3,0,7,8,1,13] involves the full multiplication $2\times(4\times(3\times(0\times(7\times(8\times(1\times 13))))))$, which is not particularly perspicuous, given that 0 is absorbing for $\times$. A better try is:

$$\Pi_2 \;=\; Yf.\lambda l.\; ?l \mapsto [\,1\,,\, (\mathtt{h}(l) = 0) \mapsto [\,0\,,\, \mathtt{h}(l) \times f(\mathtt{t}(l))\,]\,]$$

Here, the final multiplications by 7, 8, 1, and 13 have been avoided. But the execution still involves the successive multiplications of 0 by 3, 4, and 2. The following program, which makes use of the control operator κ, takes care of this:

$$\Pi_3 \;=\; \kappa k.Yf.\lambda l.\; ?l \mapsto [\,1\,,\, (\mathtt{h}(l) = 0) \mapsto [\,k\,(\lambda l' 0)\,,\, \mathtt{h}(l) \times f(\mathtt{t}(l))\,]\,]$$

It is easily checked that the execution on the same input [2,4,3,0,7,8,1,13] now returns 0 without performing any multiplication.

Remark 4. We can reach the same goal (of avoiding any multiplication by 0) using CPS (cf. Remark 3). The CPS tail-recursive version of Π_2 is:

$$\Pi_4 \;=\; Yf.\lambda l\lambda k''.\; ?l \mapsto [\,k''\,1\,,\, (\mathtt{h}(l) = 0) \mapsto [\,k''\,0\,,\, f(\mathtt{t}(l))(\lambda x.k''(\mathtt{h}(l) \times x))\,]\,]$$

It should be clear how tail-recursiveness is achieved: the additional parameter k'' is an abstraction of the stack/context. If k'' currently stands for E, then $\lambda x.k''(\mathtt{h}(l) \times x)$ stands for $E[\mathtt{h}(l) \times []]$. The program Π_4 does no better than Π_2, as it does not avoid to multiply by zero back along the recursive calls. But the following program Π_5 avoids this:

$$\Pi_5 \;=\; Yf.\lambda l\lambda k'.\; ?l \mapsto [\,k'\,1\,,\, (\mathtt{h}(l) = 0) \mapsto [\,0\,,\, f(\mathtt{t}(l))(\lambda x.k'(\mathtt{h}(l) \times x))\,]\,]$$

We owe to Olivier Danvy the following rationale for a smooth transformation from Π_4 to Π_5. The program Π_4 takes a list and a function from `nat` (the type of natural numbers) to `nat` as arguments and returns a natural number. Now, natural numbers split into 0 and (strictly) positive numbers, let us write this as $\mathtt{nat} = 0 + \mathtt{nat}^\star$. There is a well-known isomorphism between $(A + B) \to C$ and $(A \to C) \times (B \to C)$. By all this, we can rewrite Π_4 as

$$\Pi_5' \;=\; Yf.\lambda l\lambda k\lambda k'.\; ?l \mapsto [\,k'\,1\,,\, (\mathtt{h}(l) = 0) \mapsto [\,k\,0\,,\, f(\mathtt{t}(l))(\lambda x.k'(\mathtt{h}(l) \times x))\,]\,]$$

(with $k : 0 \to \mathtt{nat}$ and $k' : \mathtt{nat}^\star \to \mathtt{nat}$, where (k, k') represents $k'' : \mathtt{nat} \to \mathtt{nat}$). We then remark that k is not modified along the recursive calls, hence there is no need to carry it around. Assuming that k was initially mapping 0 to 0, we obtain Π_5. So, the CPS program Π_5 gets rid of k and retains only k'. Quite dually, we could say that the program Π_3 gets rid of k' (which has the normal control behaviour) and retains only k (whose exceptional control behaviour is handled via the κ abstraction).

A similar use of κ abstraction leads to a more "natural" way of programming the function underlying program F of section 1:

$$F' \;=\; \lambda n.\lambda l.\kappa k.(Y\, f.\lambda l_1.?l_1 \mapsto [\, nil\,,\, (\mathtt{h}(l_1) = n) \mapsto [\, k(f(\mathtt{t}(l_1)))\,,\, \mathtt{h}(l_1)\cdot f(\mathtt{t}(l_1))\,]\,])\, l$$

We set $\epsilon' \;=\; Y\, f.\lambda l_1.?l_1 \mapsto [\, nil\,,\, (\mathtt{h}(l_1) = 3) \mapsto [\, \star_{[]}(f(\mathtt{t}(l_1)))\,,\, \mathtt{h}(l_1)\cdot f(\mathtt{t}(l_1))\,]\,])\, l$, and we abbreviate $4 \cdot nil$ as 4. Here is the execution of F' on the same input as above:

$$\begin{aligned}
\langle\, F'(3)(1\cdot(3\cdot(2\cdot(3\cdot 4)))) \parallel [\,]\,\rangle &\to^\star \langle\, \epsilon'\,(1\cdot(3\cdot(2\cdot(3\cdot 4)))) \parallel [\,]\,\rangle \\
&\to^\star \langle\, \epsilon'\,(3\cdot(2\cdot(3\cdot 4))) \parallel 1\cdot[\,]\,\rangle \\
&\to^\star \langle\, \star_{[]}(\epsilon'(2\cdot(3\cdot 4))) \parallel 1\cdot[\,]\,\rangle \\
&\to^\star \langle\, \epsilon'(2\cdot(3\cdot 4)) \parallel [\,]\,\rangle \\
&\to^\star \langle\, 4 \parallel [\,]\,\rangle
\end{aligned}$$

Exercise 3. [6] Consider a slight variation of the toy language, in which lists are replaced by binary trees whose leaves are labeled by integers. This is achieved by reusing the operations $\cdot$, h, t, ?, and by removing *nil*: a tree t is either a number or is of the form $t_1 \cdot t_2$; the meaning of h and t are "left immediate subtree" and "right immediate subtree", respectively; $?t$ is now a function from trees to a sum type whose values are F or integers, it returns F if $t = t_1 \cdot t_2$ and n if $t = n$. The weight of a tree is computed as follows: $w(n) = n$, and $w(t_1 \cdot t_2) = w(t_1) + w(t_2) + 1$. A tree is called well-balanced if $t = n$, or if $t = t_1 \cdot t_2$ and $w(t_1) = w(t_2)$ and t_1, t_2 are well-balanced. Write three programs for testing if a tree is well-balanced. All programs should traverse the input tree only once. The second program should save on weight computations, the third one should also save on successive returns of the negative information that the tree is not well-balanced. (Hint: for the first two programs, make use of the above sum type.)

So far, we have only demonstrated how the κ construct allows us to escape from an evaluation context. The following exercises propose examples where continuations are passed around in more sophisticated ways. Exercises 5 and 6 are variations on the theme of *coroutines*. Coroutines are two programs that are designed to be executed in an interleaving mode, each with its own stack of execution. Each program works in turn for a while until it calls the other. Call them P and Q, respectively. When P calls Q, the execution of P is suspended until P is called back by Q, and then P resumes its execution in the context it had reached at the time of its last suspension.

Exercise 4. [11, 12] The following programs illustrate the reuse of evaluation contexts or continuations. What do they compute?

$$(\kappa k.\lambda x.k(\lambda y.x + y))\, 6$$
$$\kappa l.(\lambda(a,h).h(a+7))(\tau(3,l))\ (\tau = \lambda(n,p).\kappa k.(\lambda m.k(m,p))(\kappa q.k(n,q)))$$

Exercise 5. [12] Consider the following programs:

$$\pi = \lambda a.\phi(\lambda x.\, write\, a; x) \quad \text{and} \quad \phi = \lambda f.\lambda h.\kappa k.h(fk)$$

where the new command *write* applies to a character string, say, $'toto$, and is executed as follows:

$$\langle\, write\, 'toto; M \mid E \,\rangle \;\rightarrow\; \langle\, M \mid E \,\rangle\, !toto$$

by which we mean that the execution prints or displays *toto* and then proceeds. Describe the execution in call-by-name of $((\pi\, 'ping)(\pi\, 'pong))((\pi\, 'ping)(\pi\, 'pong))$. Does it terminate? Which successive strings are printed? (Hint: setting $P = (\pi\, 'ping)(\pi\, 'pong)$, $V_a = \lambda h.\kappa k.h((\lambda x..\, write\, a; x)k)$, and $E = [\,]P$, here are some intermediate steps:

$$\begin{aligned}\langle\, PP \mid [\,] \,\rangle &\rightarrow^\star \langle\, V_{'pong} \mid E[V_{'ping}[\,]] \,\rangle \\ &\rightarrow^\star \langle\, \star_E \mid E[[]((\lambda x..\, write\, 'pong; x)\star_E)] \,\rangle \\ &\rightarrow \langle\, (\lambda x..\, write\, 'pong; x)\star_E \mid E \,\rangle.)\end{aligned}$$

Exercise 6. [8] The toy language is extended with references and commands:

$$M := \cdots \mid \texttt{let}\, x \,=\, \texttt{ref}\, V\, \texttt{in}\, M \mid !x \mid x := V \mid M; M$$

(a variable defined wih the `ref` construct is called a reference). The machine states have now a store component (a list S of term/reference associations), notation $\langle\, M \mid E \,\rangle_S$. The evaluation rules are as follows:

$\langle\, M \mid E \,\rangle_S \;\rightarrow\; \langle\, M' \mid E' \,\rangle_S$ (for all the above rules $\langle\, M \mid E \,\rangle \rightarrow \langle\, M' \mid E' \,\rangle$
$\langle\, \texttt{let}\, x \,=\, \texttt{ref}\, V\, \texttt{in}\, N \mid E \,\rangle_S \;\rightarrow\; \langle\, N \mid E \,\rangle_{S[x \leftarrow V]}$ (x not defined in S)
$\langle\, !x \mid E \,\rangle_S \;\rightarrow\; \langle\, S(x) \mid E \,\rangle_S$
$\langle\, x := V \mid E \,\rangle_S \;\rightarrow\; \langle \mid E \,\rangle_{S[x \leftarrow V]}$
$\langle\, M; N \mid E \,\rangle_S \;\rightarrow\; \langle\, M \mid E[[]; N] \,\rangle_S$
$\langle \mid E[[]; N] \,\rangle_S \;\rightarrow\; \langle\, N \mid E[] \,\rangle_S$

Write two programs `get_first` and `get_next` that work on a binary tree (cf. exercise 2) and whose effects are the following: (`get_first` t) returns the value of the first leave (in left-to-right traversal) of t, and then evaluations of `get_next` return the values of the leaves of t in turn, suspending the traversal between two `get_next` calls. (Hints: define two references and two auxiliary functions `start` and `suspend` that use κ abstraction to store the stack in the respective references: `start` is invoked by `get_first` and `get_next` at the beginning (storing the context of the caller who communicates with the tree via these `get` functions), while `suspend` is invoked when the value of a leaf is returned (storing the context that will guide the search for the next leave).)

3 Curry-Howard

In this section, we assign typing judgements to terms M, to contexts E, and to commands $c = \langle\, M \mid E \,\rangle$. We restrict our attention to λ-calculus, extended with

the above control operations. We also switch to call-by-name, for simplicity (cf. remark 1). In this section, we adopt the alternative notation $M \cdot E$ for $E[[\,]M]$ ("push M on top of E considered as a stack"). The resulting syntax is as follows:

$$M ::= x \mid MN \mid \lambda x.M \mid \kappa k.M \mid \star_E$$
$$E ::= [\,] \mid M \cdot E$$

The abstract machine for this restricted language, called $\lambda\kappa$-calculus [2], boils down to the following four rules:

$$\langle\, MN \mid E \,\rangle \;\to\; \langle\, M \mid N \cdot E \,\rangle$$
$$\langle\, \lambda x.M \mid N \cdot E \,\rangle \;\to\; \langle\, M[x \leftarrow N] \mid E \,\rangle$$
$$\langle\, \kappa k.M \mid E \,\rangle \;\to\; \langle\, M[k \leftarrow \star_E] \mid E \,\rangle$$
$$\langle\, \star_{E_1} \mid M \cdot E_2 \,\rangle \;\to\; \langle\, M \mid E_1 \,\rangle$$

Note that the first rule suggests that the application and the push operation are redundant. As a matter of fact, we shall remove the application from the syntax in a short while.

As is well-known, terms are usually typed through judgements $\Gamma \vdash M : A$, where Γ is a sequence $x_1 : A_1, \ldots, x_n : A_n$ of variable declarations, and where M can also be interpreted as a notation for a proof tree of the sequent $A_1, \ldots, A_n \vdash A$. Let us recall the notion of sequent, due do the logician Gentzen (for an introduction to sequent calculus, we refer to, say, [5]). A sequent is given by two lists of formulas, separated by the sign $\vdash$ (which one reads as "proves"):

$$A_1, \ldots, A_m \vdash B_1, \ldots, B_n$$

The intended meaning is: "if A_1 and ... and A_m, then B_1 or ... or B_n". The A_i's are the assumptions, and the B_i's are the conclusions. Notice that there may be several formulas on the right of $\vdash$. In sequent calculus, limiting the right hand side list to consist of exactly one formula corresponds to intuitionistic logic. As a hint as to why multiple conclusions have to do with classical reasoning, let us examine how we can derive the excluded middle $\neg A \vee A$ (the typical non-constructive tautology of classical logic) from the very innocuous axiom $A \vdash A$. First, we denote false as $\bot$, and we encode $\neg A$ as $A \to \bot$ (a simple truth-table check will convince the reader that the encoding is sensible). Then, we use the multi-conclusion facility to do a right *weakening*. Weakening means adding more assumptions, or more conclusions (or both), its validity corresponds to something like "one who can do the most can do less". And finally, one gets the excluded middle by right implication introduction (see below):

$$\frac{\dfrac{A \vdash A}{A \vdash \bot, A}}{\vdash \neg A, A}$$

As we shall see, control operators lead us outside of intuitionistic logic, so we shall adopt unconstrained sequents rightaway.

Sequents may be combined to form proofs, through a few deduction rules. Here are two of them:

$$\frac{\Gamma \mid A \vdash \Delta \quad \Gamma \vdash A \mid \Delta}{\Gamma \vdash \Delta} \qquad \frac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \to B \mid \Delta}$$

The first rule is the cut rule, that can be interpreted backwards as "proving a theorem with the help of a lemma": in order to prove Δ from assumptions Γ, we first prove an auxiliary property A, and then prove Δ, taking the auxiliary property as an additional assumption. The second rule is the one corresponding to λ-abstraction. Read $A \to B$ as a function type. Then a program of type B depending on a parameter x of type A can be viewed as a function of type $A \to B$. (The role of the vertical bars in the sequents is explained in the next paragraph.)

More generally, the Curry-Howard isomorphism says that there is a one-to-one correspondence between proofs and programs. We shall extend here the correspondence to let also contexts and commands fit into it. We shall consider three sorts of sequents, corresponding to terms, contexts, and commands, respectively. They are given in the following table:

$$\left.\begin{array}{r} \ldots, A_i, \ldots \vdash B \mid \ldots, B_j, \ldots \\ \ldots, A_i, \ldots \mid A \vdash \ldots, B_j, \ldots \\ \ldots, A_i, , \ldots \vdash \ldots, B_j, \ldots \end{array}\right\} \longleftrightarrow \left\{\begin{array}{l} \ldots, x_i : A_i, \ldots \vdash M : B \mid \ldots, \alpha_j : B_j, \ldots \\ \ldots, x_i : A_i, \ldots \mid E : A \vdash \ldots, \alpha_j : B_j, \ldots \\ c : (\ldots, x_i : A_i, \ldots \vdash \ldots, \alpha_j : B_j, \ldots) \end{array}\right.$$

In the sequents corresponding to terms, one conclusion is singled out as the current one, and is placed between the $\vdash$ and the vertical bar. Symmetrically, in the sequents corresponding to contexts, one assumption is singled out as the current one, and is placed between the vertical bar and the $\vdash$. Note that a context is typed *on the left*: what we type is the hole of the context, that standds for the input it is wainting for. A command is obtained by cutting a conclusion that is singled out against an assumption that is singled out, and the resulting sequent has no conclusion nor assumption singled out.

We now turn to the typing rules. But we first note that the evaluation rule for κ is rather complicated: it involves copying the context, and transforming one of the copies into a term. It turns out that both $\kappa k.M$ and $\star_E$ can be encoded simply using a more primitive operation: Parigot's μ-abstraction [9], which has the following behaviour:

$$\langle\, \mu\alpha.c \parallel E \,\rangle \;\to\; c[\alpha \leftarrow E]$$

Moreover, the application can also be encoded with the help of the μ abstraction. The encodings are as follows (cf. also [3]):

$$\begin{array}{ll} \star_E = \lambda x.\mu\alpha.\langle\, x \parallel E \,\rangle & (\alpha \text{ not free in } E) \\ \kappa k.M = \mu\beta.\langle\, \lambda k.M \parallel \star_\beta \cdot \beta \,\rangle & (\beta \text{ not free in } M) \\ MN = \mu\alpha.\langle\, M \parallel N \cdot \alpha \,\rangle & (\alpha \text{ not free in } M, N) \end{array}$$

Exercise 7. Check that the encodings simulate the evaluation rules for $\star_E$, $\kappa k.M$, and MN.

Exercise 8. Prove the following equality:

$$(\kappa k.M)N = \kappa k'.M[k \leftarrow \lambda m.k'(mN)]N$$

More precisely, prove that the encodings of the two sides of this equality have a common reduct.

Note that the μ operation involves an explicit continuation variable (that may be bound to a context), while κ does not (it involves an ordinary variable that may be bound to a term representing a context). We shall give typing rules for the following more primitive syntax, called $\overline{\lambda}\mu$-calculus [1]:

$$\begin{array}{l} M ::= x \mid \lambda x.M \mid \mu\alpha.c \\ E ::= \alpha \mid M \cdot E \\ c ::= \langle\, M \mid E \,\rangle \end{array}$$

with the following reduction rules:

$$\begin{array}{l} \langle\, \lambda x.M \mid N \cdot E \,\rangle \;\rightarrow\; \langle\, M[x \leftarrow N] \mid E \,\rangle \\ \langle\, \mu\alpha.c \mid E \,\rangle \;\rightarrow\; c[\alpha \leftarrow E] \end{array}$$

Note that in addition to the ordinary variables $(x, y, k, k', \ldots)$, there are now first-class continuation variables α, in place of the (constant) empty context (the top-level continuation). We can now revisit the three sorts of typing judgements. All judgements allow us to type an expression containing free ordinary variables $\ldots, x_i : A_i, \ldots$ and continuation variables $\ldots, \alpha_j : B_j, \ldots$. The judgements $\ldots, x_i : A_i, \ldots \vdash M : B \mid \ldots, \alpha_j : B_j, \ldots$, $\ldots, x_i : A_i, \ldots \mid E : A \vdash \ldots, \alpha_j : B_j, \ldots$, and $c : (\ldots, x_i : A_i, \ldots \vdash \ldots, \alpha_j : B_j, \ldots)$ say that M delivers a value of type B, that E accepts a term of type A, and that c is a well-formed command, respectively. The typing rules are as follows:

$$\frac{}{\Gamma \mid \alpha : A \vdash \alpha : A, \Delta} \qquad \frac{}{\Gamma, x : A \vdash x : A \mid \Delta}$$

$$\frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \mid E : B \vdash \Delta}{\Gamma \mid (M \cdot E) : A \to B \vdash \Delta} \qquad \frac{\Gamma, x : A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}$$

$$\frac{c : (\Gamma \vdash \beta : B\,,\ \Delta)}{\Gamma \vdash \mu\beta.c : B \mid \Delta} \qquad \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \mid E : A \vdash \Delta}{\langle\, M \mid E \,\rangle : (\Gamma \vdash \Delta)}$$

Let us read the rules logically. The first two rules are variations of the axiom: a sequent holds if one formula A is both among the assumptions and the conclusions. The following two rules correspond to the introduction of the implication on the right and on the left: this is typical of sequent calculus style. Let us spell

out the left introduction rule, returning to the old notation $E[[\,]M]$. This expression has two bracketings $[\,]$: call them the inner hole and the outer hole. If M has type A, then the inner hole (which is the hole of $E[[\,]M]$) must have a type $A \to B$, hence $[\,]M$ has type B, and the outer hole (which is the hole of E) must have type B. Thus, sequent calculus' left introduction is interpreted as "push M on stacke E". The rule for μ can be viewed as a sort of coercion: the sequent to be proved does not vary, but the status of the sequent changes from having no assumption or conclusion singled out to having one conclusion singled out, which is a key step in writing a cut. The final rule is the cut rule: $\langle\, M \parallel E \,\rangle$ is well-formed when M has the type that E expects.

Remark 5 (for the logically oriented reader). In this typing system, we have left contraction (e.g., from $\Gamma, A, A \vdash \Delta$ deduce $\Gamma, A \vdash \Delta$) and weakening implicit: weakening is built-in in the two axioms for variables and continuation variables (when Γ or Δ or both are non-empty), and contraction is implicit in the "push" rule ($M \cdot E$) and in the cut rule ($\langle\, M \parallel E \,\rangle$).

Beyond the particularity of having a conclusion or an assumption singled out, the above rules are nothing but the rules of sequent calculus, and the above encoding of application is the essence of the translation from natural deduction style to sequent calculus style [5].

Exercise 9. Give a technical contents to the second part of remark 5, by defining a translation from λ-calculus to $\overline{\lambda}\mu$-calculus that preserves reductions. (Note that in the $\overline{\lambda}\mu$-calculus, the evaluation rules are not deterministic anymore: since commands are recursively part of the syntax, it makes sense to reduce not only at the root.)

Now we can derive the typing rules for $\kappa k.M$ and $\star_E$:

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash x : A \mid \Delta \qquad \Gamma \mid E : \boldsymbol{A} \vdash \Delta}{\langle\, x \parallel E \,\rangle : (\Gamma, x : A \vdash \Delta)}}{\Gamma, x : A \vdash \mu\alpha.\langle\, x \parallel E \,\rangle : R \mid \Delta}}{\Gamma \vdash \star_E : \boldsymbol{A} \to \boldsymbol{R} \mid \Delta}$$

Here, R is an arbitrary (fixed) formula/type of *results*. Note that we have slightly departed from the typing rules as written above, in order to make the essential use of weakening explicit: $\alpha : R$ is a fresh variable.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \,|\, \beta : A \vdash \Delta, \beta : A}}{\Gamma \vdash \star_\beta : A \to R \,|\, \Delta, \beta : A} \quad \Gamma \,|\, \beta : A \vdash \Delta, \beta : A}{\Gamma \,|\, \star_\beta \cdot \beta : (A \to R) \to A \vdash \Delta, \beta : A} \qquad \dfrac{\Gamma, k : \boldsymbol{A} \to \boldsymbol{R} \vdash M : \boldsymbol{A} \,|\, \Delta}{\Gamma \vdash \lambda k.M : (A \to R) \to A \,|\, \Delta}$$

$$\dfrac{\langle\, \lambda k.M \parallel \star_\beta \cdot \beta \,\rangle : (\Gamma \vdash \Delta, \beta : A)}{\Gamma \vdash \kappa k.M : \boldsymbol{A} \,|\, \Delta}$$

The last derivation reveals one of these unexpected mysteries that makes research so fascinating. The control feature encoded by κ abstraction corresponds under the Curry-Howard correspondence to reasoning by contradiction, as first discovered in [7]. Indeed, think of R as $\bot$. Then $A \to R$ is $\neg A$, and

$$\dfrac{\Gamma, k : \boldsymbol{A} \to \boldsymbol{R} \vdash M : \boldsymbol{A} \,|\, \Delta}{\Gamma \vdash \kappa k.M : \boldsymbol{A} \,|\, \Delta}$$

reads as: "if we can prove A assuming $\neg A$, then we reach a contradiction, and hence A is proved". The implication $((A \to R) \to A) \to A$ is known as Peirce's law. The reader will find related classical reasoning principles in the following exercise.

Exercise 10. We call the sequents $\neg\neg A \vdash A$ and $\bot \vdash A$ double negation elimination and $\bot$ elimination, respectively.

(1) Show that Peirce's law plus $\bot$ elimination imply double negation elimination (hint: apply the contravariance of implication, i.e., if A' implies A, then $A \to B$ implies $A' \to B$).

(2) Show that double negation elimination implies $\bot$ elimination (hint: prove that $\bot$ implies $\neg\neg B$).

(3) Show that double negation elimination implies Peirce's law (hint: use (2)).

Remark 6. Double negation elimination (cf. exercise 10) corresponds to another control operator, Felleisen's $\mathcal{C}$, whose behaviour is the following:

$$\langle\, \mathcal{C}(M) \parallel E \,\rangle \;\to\; \langle\, M \parallel \star_E \cdot [\,] \,\rangle$$

Thus, $\mathcal{C}(\lambda k.M)$ is quite similar to $\kappa k.M$, except that the stack is not copied, but only captured. The $\lambda\mu$ counterpart of $\mathcal{C}(M)$ is given by $\mu\beta.\langle\, M \parallel \star_\beta \cdot \alpha \,\rangle$ where the variables β and α are not free in M; α can be understood as a name for the toplevel continuation. The typing, as literally induced by the encoding, is as follows

$$\dfrac{\Gamma \vdash M : (A \to R) \to R \,|\, \Delta}{\Gamma \vdash \mathcal{C}(M) : A \,|\, \alpha : R, \Delta}$$

It looks a little odd, because α is a variable not mentioned in the $\mathcal{C}$ construction. One way out is to assimilate R with $\bot$, which amounts to viewing R as the (unique) type of *final* results. Then we can remove altogether $\alpha : \bot$ from the judgement (as "Δ or $\bot$" is the same as Δ), and obtain:

$$\frac{\Gamma \vdash M : (A \to \bot) \to \bot \mid \Delta}{\Gamma \vdash \mathcal{C}(M) : A \mid \Delta}$$

i.e., "$\mathcal{C}$ is double negation elimination" [7].

4 Conclusion

We have shown some basic relations between continuations and control operators, abstract machines, and sequent caclulus. The connection with logic is lost when we admit recursion into the language (section 2). But the detour through logic is extremely useful, as it brings to light a deep symmetry between terms and contexts.

The $\overline{\lambda}\mu$ calculus can be extracted from the logical considerations and can then be considered as an untyped calculus *per se*. An extension of the $\overline{\lambda}\mu$-calculus that allows for a completely symmetric account of call-by-name and call-by-value is presented in [1].

References

1. Curien, P.-L., and Herbelin, H., The duality of computation, Proc. International Conference on Functional Programming 2000, IEEE (2000).
2. Danos, V., and Krivine, J.-L., Disjunctive tautologies and synchronisation schemes, Proc. Computer Science Logic 2000.
3. de Groote, Ph., On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control, in Proc. Logic Programming and Automated Reasoning '94, Lecture Notes in Computer Science 822, Springer (1994).
4. Felleisen, M., and Friedman, D., Control operators, the SECD machine, and the λ-calculus, in Formal Description of Programming Concepts III, 193-217, North Holland (1986).
5. Girard, J.-Y., Lafont, Y., and Taylor, P., Proofs and Types, Cambridge University Press (1989).
6. Greussay, P., Contribution à la définition interprétative et à l'implémentation des lambda-langages, Thèse d'Etat, Université Paris VII (1977).
7. Griffin, T., A formula-as-types notion of control, in Proc. Principles of Programming Languages 1990, IEEE (1990).
8. Gunter, C., Rémy, D., and Riecke, J., A generalization of exceptions and control in ML-like languages, in Proc. Functional Programming and Computer Architecture '95, ACM Press (1995).
9. Parigot, M. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction, Proc. of the International Conference on Logic Programming and Automated Reasoning, St. Petersburg, Lecture Notes in Computer Science 624 (1992).

10. Prawitz, D., Natural deduction, a proof-theoretical study, Almqvist & Wiskell (1965).
11. Reynolds, J., Theories of programming languages, Cambridge University Press (1998).
12. Thielecke, H., Categorical structure of continuation passing style, PhD Thesis, Univ. of Edinburgh (1997).