Lecture Notes on
The K Machine

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
Thursday, October 10, 2019

1 Introduction

After examining an exceedingly pure, but universal notion of computation in the λ-calculus, we have been building up an increasingly expressive language including recursive types. The standard theorems to validate the statics and dynamics are progress and preservation, relying also on canonical forms. We have also seen the generic principles such as recursion and exceptions can be integrated into our language elegantly, with the necessary modifications of the progress theorem. We have also seen that the supposed opposition of dynamic and static typing is instead just a reflection of breadth of properties we would like to enforce statically, and the supposed opposition of eager (strict) and lazy constructors is just a question of which types we choose to include in our language.

At this point we briefly turn our attention to defining the dynamics of the constructs at a lower level of abstraction that we have done so far. This introduces some complexity in what we call "dynamic artifacts", that is, objects beyond the source expressions that help us describe how programs execute. In this lecture, we show the K machine in which a *stack* is made explicit. This stack can also be seen as a *continuation*, capturing everything that remains to be done after the current expression has been evaluated. At the end of the lecture we show an elegant high-level implementation of the K machine in Haskell.

2 Introducing the K Machine

Let's review the dynamics of functions.

$$\frac{}{\lambda x.\, e\ \mathit{val}}\ \mathsf{val/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathsf{step/app}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}\ \mathsf{step/app}_2$$

$$\frac{}{(\lambda x.\, e_1')\ v_2 \mapsto [v_2/x]e_1'}\ \mathsf{step/beta}$$

The rules $\mathsf{step/app}_1$ and $\mathsf{step/app}_2$ are *congruence rules*: they descend into an expression e in order to find a *redex*, $(\lambda x.\, e_1')\ v_2$ in this case. The reduction rule step/beta is the "actual" computation step, which takes place when a *constructor* (here: λ-abstraction) is met by a *destructor* (here: application).

The rules for all other forms of expression follow the same pattern. The definition of a value of the given type guides which congruence rules are required. Overall, the preservation and progress theorems verify that a particular set of rules for a type constructor was defined coherently.

In a multistep computation

$$e_0 \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto e_n = v$$

each expression e_i represents *the whole program* and v its final value. This makes the dynamics economical: only expressions are required when defining it. But a straightforward implementation would have to test whether expressions are values, and also *find* the place where the next reduction should take place by traversing the expression using congruence rules.

It would be a little bit closer to an implementation if we could keep track where in a large program we currently compute. The key idea needed to make this work is to also remember *what we still have to do after we are done evaluating the current expression*. This is the role of a *continuation* (read: "*how we continue after this*"). In the particular abstract machine we present, the continuation is organized as a stack, which appears to be a natural data structure to represent the continuation.

The machine has two different forms of states

$k \triangleright e$	evaluate e with continuation k
$k \triangleleft v$	return value v to continuation k

In the second form, we will always have v *val*. We call this an *invariant* or *presupposition* and we have to verify that all transition rules of the abstract machine preserve this invariant.

As for continuations, we'll have to see what we need as we develop the dynamics of the machine. For now, we only know that we will need an *initial continuation* or *empty stack*, written as ϵ.

$$\text{Continuations} \quad k \quad ::= \quad \epsilon \mid \ldots$$

In order to evaluate an expression, we start the machine with

$$\epsilon \triangleright e$$

and we expect that it transitions to a final state

$$\epsilon \triangleleft v$$

if and only if $e \mapsto^* v$. Actually, we can immediately generalize this: no matter what the continuation k, we want evaluation of e return the value of e to k:

> *For any continuation k, expression e and value v,*
> $k \triangleright e \mapsto^* k \triangleleft v$ *iff* $e \mapsto^* v$

We should keep this in mind as we are developing the rules for the K machine.

3 Evaluating Functions

Just as for the usual dynamics, the transitions of the machine are organized by type. We begin with functions. An expression $\lambda x.\, e$ is a value. Therefore, it is immediately returned to the continuation.

$$k \triangleright \lambda x.\, e \quad \mapsto \quad k \triangleleft \lambda x.\, e$$

It is immediate that the theorem we have in mind about the machine is satisfied by this transition.

How do we evaluate an application $e_1\ e_2$? We start by evaluating e_1 until it is a value, then we evaluate e_2, and then we perform a β-reduction. When we evaluate e_1 we have to remember what remains to be done. We do this with the continuation

$$(\_\ e_2)$$

which has a blank in place of the expression that is currently being evaluated. We push this onto the stack, because once this continuation has done its work, we still need to do whatever remains after that.

$$k \triangleright e_1\, e_2 \quad \mapsto \quad k \circ (\_\ e_2) \triangleright e_1$$

When the evaluation of e_1 returns a value v_1 to the continuation $k \circ (\_\ e_2)$ we evaluate e_2 next, remembering we have to pass the result to v_1.

$$k \circ (\_\ e_2) \triangleleft v_1 \quad \mapsto \quad k \circ (v_1\ \_) \triangleright e_2$$

Finally, when the value v_2 of e_2 is returned to this continuation we can carry out the β-reduction, substituting v_2 for the formal parameter x in the body e_1' of the function. The result is an expression that we then proceed to evaluate.

$$k \circ ((\lambda x.\, e_1')\ \_) \triangleleft v_2 \quad \mapsto \quad k \triangleright [v_2/x]e_1'$$

The continuation for $[v_2/x]e_1'$ is the original continuation of the application, because the ultimate value of the application is the ultimate value of $[v_2/x]e_1'$.

Summarizing the rules pertaining to functions:

$$\begin{array}{rcllcrcl}
k & \triangleright & \lambda x.\, e & \mapsto & & k & \triangleleft & \lambda x.\, e \\
k & \triangleright & e_1\, e_2 & \mapsto & k \circ (_\ e_2) & \triangleright & e_1 \\
k \circ (_\ e_2) & \triangleleft & v_1 & \mapsto & k \circ (v_1\ _) & \triangleright & e_2 \\
k \circ ((\lambda x.\, e_1')\ _) & \triangleleft & v_2 & \mapsto & k & \triangleright & [v_2/x]e_1'
\end{array}$$

And the continuations required:

$$\begin{array}{rrcl}
\text{Continuations} & k & ::= & \epsilon \\
 & & \mid & k \circ (_\ e_2) \mid k \circ (v_1\ _)
\end{array}$$

4 A Small Example

Let's run the machine through a small example,

$$((\lambda x.\, \lambda y.\, x)\, v_1)\, v_2$$

for some arbitrary values v_1 and v_2.

$$
\begin{array}{lrcl}
 & \epsilon & \triangleright & ((\lambda x.\, \lambda y.\, x)\, v_1)\, v_2 \\
\mapsto & \epsilon \circ (\_\ v_2) & \triangleright & (\lambda x.\, \lambda y.\, x)\, v_1 \\
\mapsto & \epsilon \circ (\_\ v_2) \circ (\_\ v_1) & \triangleright & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (\_\ v_2) \circ (\_\ v_1) & \triangleleft & \lambda x.\, \lambda y.\, x \\
\mapsto & \epsilon \circ (\_\ v_2) \circ ((\lambda x.\, \lambda y.\, x)\ \_) & \triangleright & v_1 \\
\mapsto^* & \epsilon \circ (\_\ v_2) \circ ((\lambda x.\, \lambda y.\, x)\ \_) & \triangleleft & v_1 \\
\mapsto & \epsilon \circ (\_\ v_2) & \triangleright & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ (\_\ v_2) & \triangleleft & \lambda y.\, v_1 \\
\mapsto & \epsilon \circ ((\lambda y.\, v_1)\ \_) & \triangleright & v_2 \\
\mapsto^* & \epsilon \circ ((\lambda y.\, v_1)\ \_) & \triangleleft & v_2 \\
\mapsto & \epsilon & \triangleright & v_1 \\
\mapsto^* & \epsilon & \triangleleft & v_1
\end{array}
$$

If v_1 and v_2 are functions, then the multistep transitions based on our desired correctness theorem are just a single step each.

We can see that the steps are quite small, but that the machine works as expected. We also see that some *values* (such as v_1) appear to be evaluated more than once. A further improvement of the machine would be to mark values so that they are not evaluated again.

5 Eager Pairs

Functions are lazy in the sense that the body of a λ-abstraction is not evaluated, even in a call-by-value language. As another example we consider eager pairs $\tau_1 \times \tau_2$. In lecture we actually did sums, but the same pattern emerges for both. Recall the rules:

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\langle v_1, v_2\rangle\ \mathit{val}}\ \mathsf{val/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle}\ \mathsf{step/pair}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle}\ \mathsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e)}\ \mathsf{step/case/pair}_0$$

$$\frac{v_1\ \mathit{val} \quad v_2\ \mathit{val}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e) \mapsto [v_1/x_1, v_2/x_2]e}\ \mathsf{step/case/pair}$$

We develop the rules in a similar way. Evaluation of a pair begins by evaluating the first component.

$$k \triangleright \langle e_1, e_2 \rangle \quad \mapsto \quad k \circ \langle \_, e_2 \rangle \triangleright e_1$$

When the value is returned, we start with the second component.

$$k \circ \langle \_, e_2 \rangle \triangleleft v_1 \quad \mapsto \quad k \circ \langle v_1, \_ \rangle \triangleright e_2$$

When the second value is returned, we can immediately form the pair (a new value) and return it to the continuation further up the stack.

$$k \circ \langle v_1, \_ \rangle \triangleleft v_2 \quad \mapsto \quad k \triangleleft \langle v_1, v_2 \rangle$$

For a case expression, we need to evaluate the subject of the case.

$$k \triangleright \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e) \quad \mapsto \quad k \circ \mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleright e_0$$

When e_0 has been evaluated, a pair should be returned to this continuation, and we can carry out the reduction and continue with evaluating e after substitution.

$$k \circ \mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleleft \langle v_1, v_2 \rangle \quad \mapsto \quad k \triangleright [v_1/x_1, v_2/x_2]e$$

To summarize:

$$\begin{array}{rcl}
k \triangleright \langle e_1, e_2 \rangle & \mapsto & k \circ \langle _, e_2 \rangle \triangleright e_1 \\
k \circ \langle _, e_2 \rangle \triangleleft v_1 & \mapsto & k \circ \langle v_1, _ \rangle \triangleright e_2 \\
k \circ \langle v_1, _ \rangle \triangleleft v_2 & \mapsto & k \triangleleft \langle v_1, v_2 \rangle \\
k \triangleright \mathsf{case}\ e_0\ (\langle x_1, x_2 \rangle \Rightarrow e) & \mapsto & k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleright e_0 \\
k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) \triangleleft \langle v_1, v_2 \rangle & \mapsto & k \triangleright [v_1/x_1, v_2/x_2]e
\end{array}$$

$$\begin{array}{rcll}
\text{Continuations} \quad k & ::= & \epsilon & \\
& \mid & k \circ (_\ e_2) \mid k \circ (v_1\ _) & (\rightarrow) \\
& \mid & k \circ \langle _, e_2 \rangle \mid k \circ \langle v_1, _ \rangle \mid k \circ \mathsf{case}\ _\ (\langle x_1, x_2 \rangle \Rightarrow e) & (\times)
\end{array}$$

6 Typing the K Machine

We postpone a correctness proof for the K machine to the beginning of next lecture. For now, we study the statics of the machine.

In general, it is informative to maintain static typing to the extent possible when we transform the dynamics. If there is a new language involved we might say we have a *typed intermediate language*, but even if in the case of the

K machine where we still evaluate expressions and just add continuations, we still want to maintain typing.

We type a continuation as *receiving* a value of type τ and eventually producing the final answer for the whole program of type σ. That is, $k \div \tau \Rightarrow \sigma$. Continuations are always closed, so there is no context Γ of free variables. We use a different symbol $\div$ for typing and $\Rightarrow$ for the functional interpretation of the continuation so there is no confusion with the usual notation.

The easiest case is

$$\frac{}{\epsilon \div \tau \Rightarrow \tau}$$

since the empty continuation ϵ immediately produces the value that it is passed as the final value of the computation.

We consider $k \circ (\_\ e_2)$ in some detail. This is a continuation that takes a value of type $\tau_2 \to \tau_1$ and applies it to an expression $e_2 : \tau_2$. The resulting value is passed to the remaining continuation k. The final answer type of $k \circ (\_\ e_2)$ and k are the same σ. Writing this out in the form of an inference rule:

$$\frac{k \div \tau_1 \Rightarrow \sigma \quad \cdot \vdash e_2 : \tau_2}{k \circ (\_\ e_2) \div (\tau_2 \to \tau_1) \Rightarrow \sigma}$$

The order in which we develop this rule is important: when designing or recalling such rules yourself we strongly recommend you fill in the various judgments and types incrementally, as we did in lecture.

The other function-related continuations follows a similar pattern. We arrive at

$$\frac{k \div \tau_1 \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_2 \to \tau_1 \quad v_1\ \mathit{val}}{k \circ (v_1\ \_) \div \tau_2 \Rightarrow \sigma}$$

Pairs follow a similar pattern and we just show the rules.

$$\frac{k \div (\tau_1 \times \tau_2) \Rightarrow \sigma \quad \cdot \vdash e_2 : \tau_2}{k \circ \langle \_, e_2 \rangle \div \tau_1 \Rightarrow \sigma} \qquad \frac{k \div (\tau_1 \times \tau_2) \Rightarrow \sigma \quad \cdot \vdash v_1 : \tau_1 \quad v_1\ \mathit{val}}{k \circ \langle v_1, \_ \rangle \div \tau_2 \Rightarrow \sigma}$$

$$\frac{k \div \tau' \Rightarrow \sigma \quad x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau'}{k \circ \mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e') \div (\tau_1 \times \tau_2) \Rightarrow \sigma}$$

With these rules, we can state preservation and progress theorems for the K machine, but their formulation and proof entirely follow previous developments so we elide them here.

7 Implementing the K Machine

The K machine can be extended to encompass all the type constructors we have introduced so far. Both statics and dynamics (almost) write themselves, following the same ideas we have presented in this lecture. During lecture, we also live-coded an elegant implementation of the K-machine, adding the unit type 1 for good measure.

The first question is how to implement the source expressions. We use a deep embedding in the sense that both constructors and destructors of each type have an explicit representation. But we nevertheless use functions in the metalanguage to represent bound variables together with their scope in the object language, a technique called *higher-order abstract syntax*. In the textbook, at the level of mathematical discourse, expressions with bindings are represented as *abstract binding trees*.

In Haskell, we write

```
data E = Lam (E -> E)
       | App E E
       | Pair E E
       | CasePair E (E -> E -> E)
       | Unit
       | CaseUnit E E
```

Note that λ-abstraction binds one variable and the case construct over pairs binds two.

The second question is how we represent the continuation stack. The idea suggested by the analysis in the previous section is that the continuation stack itself might be represented as a function. We represent $k \triangleright e$ by *eval* e k and $k \triangleleft v$ by *retn* v k. Writing the continuation as a second argument aids in the readability of the code.

```
eval :: E -> (E -> E) -> E
retn :: E -> (E -> E) -> E
```

Now we transcribe the rules. For example,

$$k \triangleright \lambda x.\, e \;\; \mapsto \;\; k \triangleleft \lambda x.\, e$$

Since a λ-expression is a value, evaluating it immediately returns it to the continuation. This becomes

```
eval (Lam f) k = retn (Lam f) k
```

Also, returning a value to a continuation simply applies the continuation (which is a function) to the value.

```
retn v k = k v
```

Application $e_1\ e_2$ is a bit more complicated. First, we evaluate e_1, returning its value to the continuation.

```
eval (App e1 e2) k = eval e1 (\v1 -> ...)
```

The continuation (here `...`) that expects v_1 has to evaluate e_2 next and pass *its* value to a further continuation.

```
eval (App e1 e2) k = eval e1 (\v1 -> eval e2 (\v2 -> ...))
```

Now we have to perform the actual reduction, substituting v_2 in the body of the λ-expression that is v_1. In order to be able to write that, we pattern-match against a λ-value when we receive v_1.

```
eval (App e1 e2) k = eval e1 (\(Lam f) -> eval e2 (\v2 -> ...))
```

Since the constructor `Lam :: (E -> E) -> E`, we see that `f :: E -> E`. Applying `f` to `e2` will effectively substitute `e2` into the body of `f`.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> ... (f v2) ...))
```

That will result in an expression representing $[v_2/x]e_1'$, which we need to evaluate further.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> eval (f v2) ...))
```

Finally, we have to pass the original continuation to this evaluation.

```
eval (App e1 e2) k =
    eval e1 (\(Lam f) -> eval e2 (\v2 -> eval (f v2) k))
```

The remaining cases in evaluation are derived from the transition rules of the abstract machine in a similar manner. We do not make continuations or stacks explicit as a data structure, but represent them as functions. We show the completed code.

```
data E = Lam (E -> E)
       | App E E
       | Pair E E
       | CasePair E (E -> E -> E)
       | Unit
       | CaseUnit E E

eval :: E -> (E -> E) -> E
retn :: E -> (E -> E) -> E

eval (Lam f) k = retn (Lam f) k
eval (App e1 e2) k = eval e1 (\(Lam f) ->
                              eval e2 (\v2 -> eval (f v2) k))
eval (Pair e1 e2) k = eval e1 (\v1 ->
                               eval e2 (\v2 -> retn (Pair v1 v2) k))
eval (CasePair e f) k = eval e (\(Pair v1 v2) -> eval (f v1 v2) k)
eval (Unit) k = retn (Unit) k
eval (CaseUnit e f) k = eval e (\(Unit) -> eval f k)

retn v k = k v
```

This interpreter can fail with an error because we have not implemented a type-checker. Such as error could arise because pattern-matching against `(Lam f)`, `(Pair v1 v2)`, and `(Unit)` in the cases for `App`, `CasePair`, and `CaseUnit` may fail to match the value returned *if the expression is not well-typed*. Writing a type-checker on this representation is a bit tricky, and we might discuss it at a future lecture.

A more complete implementation, including fixed points, recursive types, and sums can be found on the course schedule page. There, you can also find the live-coded file from lecture where we implemented sums so we could represent the "*and*" function on booleans and execute it.

This form of continuation-passing interpreter has been proposed by Reynolds [Rey72] as a means of language definition. The K machine can be seen as a "defunctionalization" of such a higher-order interpreter.

Exercises

Exercise 1 Extend the K Machine for the following constructs, in each case writing out new continuations as necessary and giving both stepping and typing rules.

1. Constructor and destructor for the unit type 1.

2. Constructor and destructor for the sum type $\sum_{i \in I}(i : \tau_i)$.
3. Constructor and destructor for recursive types $\rho\alpha.\,\tau$.
4. The fixed point expression fix $f.\,e$.
5. Constructor and destructors for lazy pairs $\tau_1 \& \tau_2$ (see Exercise L7.1).

References

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.