Lecture Notes on The K Machine

15-814: Types and Programming Languages
Frank Pfenning

Lecture 12
Tuesday, October 19, 2021

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 (as we will see in the next lecture) 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 our own language. This is an example of a so-called *metacircular interpreter*, a particular form of a *definitional interpreter* [Rey72] which can be seen as *defining* the dynamics to the object language.

2 Introducing the K Machine

Let's review the dynamics of functions.

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

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app_1} \qquad \frac{v_1\ \mathit{value} \quad e_2 \mapsto e_2'}{v_1\, e_2 \mapsto v_1\, e_2'}\ \mathsf{step/app_2}$$

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

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 $\mathsf{step/beta}$ 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 *value*. 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

$$\epsilon \rhd e$$

and we expect that it transitions to a final state

$$\epsilon \lhd v$$

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.

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

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.

$$k \circ (\_\; e_2) \lhd v_1 \quad \mapsto \quad k \circ (v_1\; \_) \rhd e_2$$

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.

$$k \circ ((\lambda x.\, e_1')\; \_) \lhd v_2 \quad \mapsto \quad k \rhd [v_2/x]e_1'$$

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:

$$\begin{array}{rcl@{\quad}c@{\quad}rcl}
k & \rhd & \lambda x.\, e & \mapsto & k & \lhd & \lambda x.\, e \\
k & \rhd & e_1\, e_2 & \mapsto & k \circ (\_\; e_2) & \rhd & e_1 \\
k \circ (\_\; e_2) & \lhd & v_1 & \mapsto & k \circ (v_1\; \_) & \rhd & e_2 \\
k \circ ((\lambda x.\, e_1')\; \_) & \lhd & v_2 & \mapsto & k & \rhd & [v_2/x]e_1'
\end{array}$$

And the continuations required:

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

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 Sums

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 sums $\sum_{i\in I}(i : \tau_i)$. Let's recall the key rules.

$$\frac{(k \in I) \quad \Gamma \vdash e_k : \tau_k \quad \Gamma \vdash \sum_{i\in I}(i : \tau_i)\ \mathit{type}}{\Gamma \vdash k \cdot e_k : \sum_{i\in I}(i : \tau_i)}\ \mathsf{tp/tag}$$

$$\frac{\Gamma \vdash e : \sum_{i\in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e_i : \sigma \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i\in I} : \sigma}\ \mathsf{tp/cases}$$

$$\frac{e\ \mathit{value}}{k \cdot e\ \mathit{value}}\ \mathsf{val/tag} \qquad \frac{v_k\ \mathit{value}}{\mathsf{case}\ k \cdot v_k\ (i \cdot x_i \Rightarrow e_i)_{i\in I} \mapsto [v_k/x_k]e_k}\ \mathsf{step/cases/tag}$$

$$\frac{e_1 \mapsto e_1'}{k \cdot e_1 \mapsto k \cdot e_1'}\ \mathsf{step/tag} \qquad \frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (i \cdot x_i \Rightarrow e_i)_{i\in I} \mapsto \mathsf{case}\ e_0'\ (i \cdot x_i \Rightarrow e_i)_{i\in I}}\ \mathsf{step/cases}_0$$

We develop the rules in a similar manner to functions. Evaluation of a tagged expression begins by evaluating underneath the tag.

$$k \triangleright j \cdot e \quad \mapsto \quad k \circ (j \cdot \_) \triangleright e$$

When the value v is returned we can tag this value with j and pass it to the continuation.

$$k \circ (j \cdot \_) \triangleleft v \quad \mapsto \quad k \triangleleft j \cdot v$$

Evaluating a case expression requires first evaluating the subject of the case.

$$k \triangleright \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i \in I} \quad \mapsto \quad k \circ (\mathsf{case}\ \_\ (i \cdot x_i \Rightarrow e_i)_{i \in I}) \triangleright e$$

Finally, when a tagged value is returned to this continuation we select the correct branch, carry out the substitution, and evaluate the results.

$$k \circ (\mathsf{case}\ \_\ (i \cdot x_i \Rightarrow e_i)_{i \in I}) \triangleleft j \cdot v_j \quad \mapsto \quad k \triangleright [v_j/x_j]e_j$$

Summarizing the rules pertaining to sums:

$$\begin{array}{rcll}
k \triangleright j \cdot e & \mapsto & k \circ (j \cdot \_) \triangleright e & (1) \\
k \circ (j \cdot \_) \triangleleft v & \mapsto & k \triangleleft j \cdot v & (2) \\
k \triangleright \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i)_{i \in I} & \mapsto & k \circ (\mathsf{case}\ \_\ (i \cdot x_i \Rightarrow e_i)_{i \in I}) \triangleright e & (3) \\
k \circ (\mathsf{case}\ \_\ (i \cdot x_i \Rightarrow e_i)_{i \in I}) \triangleleft j \cdot v_j & \mapsto & k \triangleright [v_j/x_j]e_j & (4)
\end{array}$$

A few things to note about these rules.

For one, there is a correspondence between these rules and the rules defining the dynamics. (1) corresponds to step/tag, (2) corresponds to val/tag, (3) corresponds to step/cases$_0$ and (4) corresponds to step/cases/tag.

For another, the rules in the stepping judgment for the K machine are all axioms. So a computation as a sequence of steps will be entirely flat in its structure.

Finally, if we look at the mode, just as for the previous stepping judgment we assume the left-hand side of each stepping rule is given and we have enough information to construct the right-hand side. That is, we have the mode $s^+ \quad \mapsto \quad t^-$.

The accumulated language of continuations and machine states:

$$\begin{array}{llcll}
\text{Continuations} & k & ::= & \epsilon & \\
& & \mid & k \circ (\_\ e_2) \mid k \circ (v_1\ \_) & (\rightarrow) \\
& & \mid & k \circ (j \cdot \_) \mid k \circ (\mathsf{case}\ \_\ (i \cdot x_i \Rightarrow e_i)_{i \in I}) & (+) \\
\text{States} & s & ::= & k \triangleright e \mid k \triangleleft v &
\end{array}$$

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

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

7 Implementing the K Machine

We now proceed to implement the K machine for our language within our language, using LAMBDA's concrete syntax. Because we are implementing the language within itself, this is called a *metacircular interpreter*. We need to be careful to distinguish the *metalanguage* in which we write our interpreter from the *object language* in which should be able to execute programs.

As a matter of convenience and readability (but not a matter of essence), we will use varyadic sums in the metalanguage, and binary sums in the object language.

In these notes we only show the cases for functions $\tau_1 \to \tau_2$ and sums $\tau_1 + \tau_2$ in the object language; the other cases follow the same patterns and pose only minor challenges.

The first beautiful idea of the metacircular interpreter is to implement object-language variables by meta-language variables. This means that object-level functions are implemented via meta-level functions, but at different types. As a result, we will not need to implement *substitution*, because applying the meta-level function will have the effect of implementing object-level substitution.

But first the type *E* of object-level expressions. It is a (recursive) sum type where each constructor and destructor has a separate summand. We start with just functions.

```
type E = $E. ('lam : E -> E) + ('app : E * E)
```

There is no case for variables, since they are represented by meta-language variable. For example, using $\ulcorner\cdot\urcorner$ for the representation function for expression in our language, we have

$$\begin{array}{lcl} E & = & \mu E.\,(\mathbf{lam} : E \to E) + (\mathbf{app} : E \times E) \\ \ulcorner\lambda x.\, e\urcorner & = & \mathsf{fold}\ \mathbf{lam}\cdot(\lambda x.\ulcorner e\urcorner) \\ \ulcorner x\urcorner & = & x \\ \ulcorner e_1\, e_2\urcorner & = & \mathsf{fold}\ \mathbf{app}\cdot\langle\ulcorner e_1\urcorner,\ulcorner e_2\urcorner\rangle \end{array}$$

Here are two examples of this representation in our language

```
decl I : E
defn I = fold 'lam (\x. x)

decl omega : E
defn omega = fold 'lam (\x. fold 'app (x, x))
```

We can now define some "boilerplate" code, namely the meta-level constructor functions. To make them more easily readable, we give them in curried form.

```
decl lam : (E -> E) -> E
decl app : E -> (E -> E)
```

```

defn lam = \f. fold 'lam f
defn app = \e1. \e2. fold 'app (e1, e2)
```

The next step is to think about the representation of the stack. We represent this as a *meta-level function* from values to values. Since we don't have a separate type of object-level values (at the moment), they are represented as expressions and it is up to us, as the meta-programmer, to ensure that they are only applied the object-level values.

The interpreter is defined by a function *eval*. It is intended to satisfy the property

$$k \triangleright e \mapsto^* k \triangleleft v \quad \text{if and only if} \quad \mathit{eval}\ \ulcorner e \urcorner\, \ulcorner k \urcorner \mapsto^* \ulcorner k \urcorner\, \ulcorner v \urcorner$$

The main function *eval* e k evaluates e and passes the value to k (instead of returning it). Its first case is fairly simple: when the expression e is a λ-abstraction, it is already a value and we return it to the continuation k. We use an underscore to complete the name `eval_` because `eval` would clash with LAMBDA's **eval** keyword.

```
decl eval_ : E -> (E -> E) -> E
defn eval_ = $eval_. \e. \k.
               case (unfold e)
                 of ( 'lam _ => k e
                    | ... )
```

The second case is that of an application $e_1\, e_2$. We first have to evaluate e_1, with a continuation that evaluates e_2 next. That is,

```
decl eval_ : E -> (E -> E) -> E
defn eval_ = $eval_. \e. \k.
               case (unfold e)
                 of ( 'lam f => k e
                    | 'app (e1,e2) =>
                         eval_ e1 (\v1. eval_ e2 (\v2. ...)))
```

The rules of the K machine dictated that once we have evaluated both e_1 and e_2 to v_1 and v_2, respectively, then $v_1 = \lambda x.\, e_1'$ and we then need to evaluate $[v_2/x]e_1'$. We accomplish this in two steps: first, we match v_1 against the representation of λ-expression. This exposes the underlying meta-level function f. We then perform the substitution by applying f to v_2.

```
decl eval_ : E -> (E -> E) -> E
defn eval_ = $eval_. \e. \k.
               case (unfold e)
                 of ( 'lam f => k e
                    | 'app (e1,e2) =>
                         eval_ e1 (\v1. eval_ e2 (\v2.
                           case (unfold v1)
                             of ( 'lam f => eval_ (f v2) k ))))
```

However, this is not the final return value. Instead we pass it to the continuation k that expects the value of $e_1\, e_2$ (which will be computed as the value of $f\, v_2$).

At the "top level" the *evaluate* function passes the initial continuation to *eval*, which is $\lambda v.\, v$ corresponding to the empty stack ϵ. An interesting property of this representation is that to some extent visibility on the object language is inherited from visibility in the metalanguage. For example,

```
decl I : E
decl K : E
decl K' : E
defn I = fold 'lam (\x. x)
defn K = fold 'lam (\x. fold 'lam (\y. x))
defn K' = fold 'lam (\x. fold 'lam (\y. y))

eval I' = eval_ (fold 'app (K',K)) (\v. v)
```

shows us

```
% eval I' = eval_ (fold 'app (K', K)) (\v. v)
% 26 evaluation steps
decl I' : E
defn I' = fold 'lam ---
% success
```

In other words, we do not see the normal form because none is computed: arbitrary closed λ-expressions are values in both the object language and metalanguage.

We also see that there is a significant overhead in the interpreter: an expression which takes 1 step to reach a normal form in the small-step dynamics takes 28 steps in the metainterpreter. Of course, we imagine the metainterpreter could be compiled, so in the end it may be efficient enough for many purposes.

LAMBDA also issues a warning for missing patterns that arise because we match the return value from the recursive call only against `'lam f`, omitting the case of applications. That's because by an invariant of our code, the expression passed to the continuation is always a value (in the object language), but the type system does not understand this fact. It is interesting to consider how to refine the type, distinguishing values and expressions, so that this error does not arise. While it is possible, it is not straightforward. We may return to this in a future lecture.

For binary sums, the same techniques apply. Key is to represent the branches of a case statement as functions from the tagged value to the result.

$$
\begin{array}{lcl}
\ulcorner \mathbf{l} \cdot e \urcorner & = & \textsf{fold}\ \textsf{\textbf{left}} \cdot \ulcorner e \urcorner \\
\ulcorner \mathbf{r} \cdot e \urcorner & = & \textsf{fold}\ \textsf{\textbf{right}} \cdot \ulcorner e \urcorner \\
\ulcorner \textbf{case}\ e\ (\mathbf{l} \cdot x_1 \Rightarrow e_1 \mid \mathbf{r} \cdot x_2 \Rightarrow e_2) \urcorner & = & \textsf{fold}\ \textsf{\textbf{cases}} \cdot \langle \ulcorner e \urcorner, \langle \lambda x_1 . \ulcorner e_1 \urcorner, \lambda x_2 . \ulcorner e_2 \urcorner \rangle \rangle
\end{array}
$$

We just show the final code.

```
type E = $E. ('lam : E -> E) + ('app : E * E)
            + ('l : E) + ('r : E) + ('cases : E * (E -> E) * (E -> E))

decl eval_ : E -> (E -> E) -> E
defn eval_ = $eval_. \e. \k.
               case (unfold e)
                 of ( 'lam f => k e
                    | 'app (e1,e2) =>
                        eval_ e1 (\v1. eval_ e2 (\v2.
                          case (unfold v1)
                            of ( 'lam f => eval_ (f v2) k )))
                    | 'l e => eval_ e (\v. k (fold 'l v))
                    | 'r e => eval_ e (\v. k (fold 'r v))
                    | 'cases (e0, bl, br) =>
                      eval_ e0 (\v0. case (unfold v0)
                                      of ('l v => eval_ (bl v) k
                                         |'r v => eval_ (br v) k)))
```

The live code can be found online at k-live.cbv. Off-line, we completed the code to include all the type constructors in our language (excluding only polymorphism). You can find this code at k.cbv.

It is now very easy to change the language semantics, for example, to make the object language call-by-name (see Exercise 2)

This style of programming is called *continuation-passing style*, which is a perfect match for the K machine. This kind of interpreter is also called a *definitional interpreter* [Rey72] since it can be seen as providing a dynamics to the object language.

8 Whither Types?

We have represented all expressions in the object language with the same type *E* in the metalanguage. This means we can evaluate even expressions which have no type, such as *omega* in our example. To complete our implementation of the object language we should also provide a object-language type-checker in the metalanguage. We may return to this in a future lecture; for now we are content to have implemented the dynamics.

A metalanguage term of type E that is not the representation of a well-typed term in the object language may lead to a runtime exception when we distinguishes cases among the result of evaluation, which happens in the implementation of every destructor (in our code here, application and case over binary sums). These meta-level case expressions are not exhaustive pattern matches, but assume the represented term (and therefore also its value) are well-typed at the object level. This is an example of an *representation invariant*, and a fairly trick one, and shows that we should not expect in general that all pattern matches be exhaustive.

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 recursive types $\mu\alpha.\,\tau$.
3. The fixed point expression $\mathsf{fix}\ f.\,e$.
4. Constructor and destructors for lazy pairs $\tau_1 \,\&\, \tau_2$.

Exercise 2 Modify the implementation of the K machine so that function calls are treated according to the call-by-name discipline.

Exercise 3 Extend the K machine for general (nested) pattern matching. Give any possible new machine states explicit, and show both the typing and stepping rules for the machine. As part of this, you will have to deal with exceptions. Consider only the simplest case where exceptions cannot be caught.

Exercise 4 Distinguish a type *V* of values from expressions so that, for example, we never accidentally pass an unevaluated expression to a continuation and that the final answer is also a value. State the types of *eval* and *retn*. How much of the interpreter do you need to rewrite to guarantee this property?

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.