Lecture Notes on
Interpreters

15-814: Types and Programming Languages
Frank Pfenning

Lecture 10
Thu Sep 25, 2025

1 Introduction

So far, we have presented computation as a judgment, focusing on single-step reduction $e \mapsto e'$ and e *value*. We now apply our insight from last lecture that some judgments have an algorithmic interpretation via bottom-up construction of derivations to develop implementations. A common term for a program that executes another program is an *interpreter*. This is often contrasted with a *compiler* that translates from one language to anther, typically with lower-level constructs.

In this lecture we develop two interpreters. The first is entirely based on the idea of interpreting judgments algorithmically. The second follows the tradition of *definitional interpreters* [Reynolds, 1972]. Both of these are implemented in LAMBDA as *meta-circular interpreters*, that is, we implement a (significant fragment of) a language in itself.

Both of these exploit the expressive power of the metalanguage in interesting ways. One cool technique is to exploit meta-level λ-abstraction to represent object-level variable binding, which is often called *higher-order abstract syntax* or *abstract binding trees*. It is a pervasive technique in logical frameworks such as LF [Harper et al., 1993]. A second cool technique is to exploit meta-level functions to capture *continuations* that embody whatever remains to be done to finish evaluation of a program.

2 An Algorithmic Evaluation Judgment

Given our intuition about the stepping and value judgments, we would expect the following modes:

$$e^{+} \mapsto e'^{-}$$
$$e^{+} \ \textit{value}$$

An interpreter should not just step, but reduce an expression all the way to a final value. So we introduce an additional judgment

$$e \hookrightarrow v$$

defined by the rules

$$\frac{e \ \textit{value}}{e \hookrightarrow e}\ \mathsf{eval/done} \qquad \frac{e \mapsto e' \quad e' \hookrightarrow v}{e \hookrightarrow v}\ \mathsf{eval/step}$$

It is easy to verify that this judgment is well-moded with respect to

$$e^+ \hookrightarrow v^-$$

assume that the stepping and value judgment have the modes shown above (as we expect). Even if easy, let's reason through the rule.

$$\frac{e^+ \; \mathit{value}}{e^+ \hookrightarrow e^-}\;\mathsf{eval/done}$$

1. We are given e as input, so we can ask the question if it is a value.
2. We can also return it as output.

And:

$$\frac{e^+ \mapsto e'^- \quad e'^+ \hookrightarrow v^-}{e^+ \hookrightarrow v^-}\;\mathsf{eval/step}$$

1. We are given e.
2. So we can ask the question in the first premise, which may give us an e'.
3. Since e' is now known, we can ask the second question and obtain a v.
4. This allows us to return v as output.

In order to translate these judgments to types, let's recall the statement of progress.

Progress. If $\cdot \vdash e : \tau$ then either $e \mapsto e'$ for some e' or e *value*.

In other words, a closed and well-typed expression either reduces or it is already a value. Assuming we have already defined a type `exp` of expressions, we can conjecture the following types for the algorithmic interpretation of the stepping and value judgments.

```
type exp = ... (omitted) ...

type outcome = ('redux : exp) + ('value : 1)

decl progress : exp -> outcome

decl eval_ : exp -> exp
```

The specification is that[1]

$$\begin{array}{ll} \mathsf{progress}\ \ulcorner e \urcorner = \mathbf{redux}\ \ulcorner e' \urcorner & \text{if } e \mapsto e' \\ \mathsf{progress}\ \ulcorner e \urcorner = \mathbf{value}\ () & \text{if } e\ \mathit{value} \end{array}$$

where $\ulcorner e \urcorner : \mathsf{exp}$ is the representation of expressions as data in the LAMBDA language. We have written "=" here, but it is really evaluation at the metalevel that reduces the left-hand side to the right-hand side. The progress theorem guarantees that this will be a total function as long as e is closed and well-typed.

Before we implement the progress function, we can already implement the eval_ function in a direct transcription of the two rules.

```
decl eval_ : exp -> exp
defn eval_ = $eval_. \e. case progress e of
             ( 'value () => e
             | 'redux e' => eval_ e')
```

[1]We write `eval_` because **`eval`** is a keyword in LAMBDA.

3 Higher-Order Abstract Syntax

In order to program the `progress` function, we first need to decide on a representation of expressions. Experience shows that the most tedious part of the implementation of progress is substitution which is needed for β-reduction. We use the following representation with tags **app** and **lam**.

$$\begin{array}{lcl} \ulcorner e_1\, e_2 \urcorner & = & \mathbf{app}(\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner) \\ \ulcorner \lambda x.\, e \urcorner & = & \mathbf{lam}(\lambda x. \ulcorner e \urcorner) \\ \ulcorner x \urcorner & = & x \end{array}$$

The strange part here is that we represent object-level variables by meta-level variables of the same name. This means that we can use function application at the meta-level to implement substitution at the object level. In other words

$$(\lambda x. \ulcorner e_2 \urcorner)\, \ulcorner e_1 \urcorner = \ulcorner [e_1/x] e_2 \urcorner$$

in the sense that the right-hand side reduces to the left-hand side in the metalanguage since $\ulcorner e \urcorner$ is a value in the metalanguage. So we take advantage of the fact that our metalanguage contains functions.

So, for the purely functional fragment we would define

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

Then, for example

```
% K = \x. \y. x
decl K : exp
defn K = fold 'lam (\x. fold 'lam (\y. x))
```

Note that the type `exp` does not have an explicit case for variables. That's because the expression is closed so that all variables occurring in it are introduced by a metalevel λ-abstraction.

With this representation in hand, we can implement the progress function.

4 From the Proof of Progress to an Implementation

The proof of the progress theorem is by rule induction on the derivation of $\cdot \vdash e : \tau$. Since the typing judgment is syntax-directed, this is implemented via cases on e, of which there are only two.

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => ...
    | 'app (e1, e2) => ...
    )
```

If **unfold**`e = 'lam f` then `e` is already a value, so we return `'value ()`.

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => ...
    )
```

The progress theorem now proceeds by applying the induction hypothesis to the typing of e_1. This manifests itself in a recursive call to progress, which could return either the redux e_1' or indicate that it is a value already.

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => case progress e1 of
                       ( 'redux e1' => ...
                       | 'value () => ...
                       )
    )
```

Recall the rule

$$\frac{e_1 \mapsto e_1'}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1$$

so the first case we just return the redux $e_1'\, e_2$.

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => case progress e1 of
                       ( 'redux e1' => 'reduce (fold 'app (e1', e2))
                       | 'value () => ...
                       )
    )
```

If e_1 is a value, we appeal to the induction hypothesis on the typing of e_2, which could either reduce or be a value. In code:

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => case progress e1 of
                       ( 'redux e1' => 'reduce (fold 'app (e1', e2))
                       | 'value () => case progress e2 of
                                      ( 'redux e2' => ...
                                      | 'value () => ...
                                      )
                       )
    )
```

If $e_2 \mapsto e_2'$ then we use the other congruence rule

$$\frac{e_1\ \mathit{value} \quad e_2 \mapsto e_2'}{e_1\, e_2' \mapsto e_1\, e_2'}\ \mathsf{step/app}_2$$

In code:

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => case progress e1 of
                       ( 'redux e1' => 'reduce (fold 'app (e1', e2))
                       | 'value () => case progress e2 of
                                      ( 'redux e2' => 'redux (fold 'app (e1, e2'))
                                      | 'value () => ...
                                      )
                       )
    )
```

The last case is that e_1 and e_2 are both values. By the canonical form theorem, e_1 must be λ-abstraction. In our case, this means it must have the form `'lam f` where `f : exp -> exp` represent the body of the abstraction as a meta-level function. We can therefore implement the rule

$$\frac{e_2 \; \mathit{value}}{(\lambda x.\, e_1')\, e_2 \mapsto [e_2/x]e_1'} \; \mathsf{beta}$$

by using function application:

```
defn progress = $progress. \e. case unfold e of
    ( 'lam f => 'value ()
    | 'app (e1, e2) => case progress e1 of
                        ( 'redux e1' => 'reduce (fold 'app (e1', e2))
                        | 'value () => case progress e2 of
                                        ( 'redux e2' => 'redux (fold 'app (e1, e2'))
                                        | 'value () => case unfold e1 of
                                                        ( 'lam f => 'reduce (f e2) )
                                        )
                        )
    )
```

In the last case we have given a case only for `'lam f` and not for application. The types of our representation cannot see that other cases are impossible, so we get a warning from the compiler. If an application were to occur at runtime, the program would terminate with an (uncatchable) exception.

We show the complete program and run it on the example of $(\lambda x.\, x)\,(\lambda x.\, x)$

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

type outcome = ('redux : exp) + ('value : 1)

decl progress : exp -> outcome

defn progress = $progress. \e. case unfold e of
      ( 'lam f => 'value ()
      | 'app (e1, e2) => case progress e1 of
                         ( 'redux e1' => 'redux (fold 'app (e1', e2))
                         | 'value () => case progress e2 of
                                        ( 'redux e2' => 'redux (fold 'app (e1, e2'))
                                        | 'value () => case unfold e1 of
                                                       ( 'lam f => 'redux (f e2) )
                                        )
                         )
      )

decl eval_ : exp -> exp
defn eval_ = $eval_. \e. case progress e of
               ( 'value () => e
               | 'redux e' => eval_ e' )

decl id : exp
defn id = fold 'lam (\x. x)

eval idid = eval_ (fold 'app (id, id))
```

The answer is only shown as

```
decl idid : exp
defn idid = fold 'lam ---
```

This is because functions in LAMBDA are not observable! Fortunately, this is the same for both meta-language and object language, so we shouldn't expect any more information.

In lecture we extended the code to also include the cases for the unit type, so we would have *some* observable value. We further explicitly included the outcome `'wrong ()`, which could be returned if the original expression is not well-typed. It is a "poor man's exception" because we have to propagate it all the case statements. According to Milner [1978], *well-typed programs cannot go wrong*, that is, they do not return `'wrong ()` (while ill-typed programs may or may not go wrong).

This code can be found in eval.cbv. It is a mostly mechanical excercise to further extend this code with other kinds of expression in the LAMBDA language (see Exercise 1).

5 A Continuation-Passing Interpreter

In this section we explore an alternative way to write an interpreter for LAMBDA. This interpreter is based on the idea of *definitional interpreters* due to Reynolds [1972]. We use the same representation of expression, but the interpreter has type

```
decl evalk : exp -> (exp -> exp) -> exp
```

where evalk $\ulcorner e \urcorner$ k evaluates e and passes the resulting value to the *continuation* k rather than returning it.

This time, we start with the language of the last section (including unit and case over unit), but we also add fixed points.

```
type exp = $exp.
             ('lam : exp -> exp)
           + ('app : exp * exp)
           + ('unit : 1)
           + ('caseu : exp * (1 -> exp))
           + ('fix : exp -> exp)
```

We start with the outline of the `evalk` function.

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
               ( 'lam f => ...
               | 'app (e1, e2) => ...
               | 'unit () => ...
               | 'caseu (e1, f) => ...
               | 'fix f => ...
               )
```

Let's look at the first branch. `'lam f` is already a value (because $\lambda x.\, e'$ *value* holds) so we just pass e to the continuation.

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
               ( 'lam f => k e
               | 'app (e1, e2) => ...
```

```
              | 'unit () => ...
              | 'caseu (e1, f) => ...
              | 'fix f => ...
              )
```

Similarly, `'unit ()` represents a value, so we pass it to k as well.

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
             ( 'lam f => k e
             | 'app (e1, e2) => ...
             | 'unit () => k e
             | 'caseu (e1, f) => ...
             | 'fix f => ...
             )
```

Application is a more interesting case. We first have to evaluate e_1 and pass its value to the continuation. That is, we start this branch as

```
| 'app (e1, e2) => evalk e1 (\v1. ...)
```

When the value of e_1 is passed to the continuation, when the have to evaluation e_2 and pass its value to a further continuation.

```
| 'app (e1, e2) => evalk e1 (\v1. evalk e1 (\v2. ...))
```

At this point, by the canonical forms theorem, we know that $v_1 = \lambda x.\, e_1'$ for some x and e_1'. On the representation, we just decompose v_1.

```
| 'app (e1, e2) => evalk e1 (\v1. evalk e1 (\v2. case unfold v1 of
                                                  ( 'lam f => ...)))
```

In higher-order abstract syntax, we have that $f : \mathsf{exp} \to \mathsf{exp}$. The substitution we need to do here is accomplished by apply f to v_2, as in the interpreter in the previous section. But what do we do with the result of the substitution? We still have to evaluate it and pass the result to the original k, because it represents the result of evaluating the application.

```
| 'app (e1, e2) => evalk e1 (\v1. evalk e1 (\v2. case unfold v1 of
                                                  ( 'lam f => evalk (f v2) k )))
```

Splicing it back this the function outline, we have so far:

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
             ( 'lam f => k e
             | 'app (e1, e2) => evalk e1 (\v1. evalk e2 (\v2.
                                  case unfold v1 of
                                  ( 'lam f => evalk (f v2) k )))
             | 'unit () => k e
             | 'caseu (e1, f) => ...
             | 'fix f => ...
             )
```

The branch for a case over a value of unit type is similar, except we have to apply f to $()$ instead of a value.

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
             ( 'lam f => k e
             | 'app (e1, e2) => evalk e1 (\v1. evalk e2 (\v2.
                                  case unfold v1 of
                                  ( 'lam f => evalk (f v2) k )))
             | 'unit () => k e
             | 'caseu (e1, f) => evalk e1 (\v1. case unfold v1 of
                                              ( 'unit () => evalk (f ()) k))
             | 'fix f => ...
             )
```

The recursive call to `evalk` evaluates the body of the case expression and passes its result to original k, because the value of the body is the value of the whole case expression.

Finally, for fixed points, we simply unroll them. We accomplish the substitution $[\textbf{fix}\ f.\, e/x]e$ by applying the function f that represents the body of the fixed point. That won't we a value in general—we still have to evaluate it.

```
decl evalk : exp -> (exp -> exp) -> exp

defn evalk = $evalk. \e. \k. case unfold e of
             ( 'lam f => k e
             | 'app (e1, e2) => evalk e1 (\v1. evalk e2 (\v2.
                                  case unfold v1 of
                                  ( 'lam f => evalk (f v2) k )))
             | 'unit () => k e
             | 'caseu (e1, f) => evalk e1 (\v1. case unfold v1 of
                                              ( 'unit () => evalk (f ()) k))
             | 'fix f => evalk (f e) k
             )
```

We still have to define top-level evaluation that calls `evalk` with a suitable continuation. For the overall computation, that is just the identity function.

```
decl eval_ : exp -> exp
defn eval_ = \e. evalk e (\v. v)
```

As an example, we can evaluate the application of the identity function to the unit element.

```
decl id : exp
defn id = fold 'lam (\x. x)

eval id1 = eval_ (fold 'app (id, fold 'unit ()))
```

We can also run the fixed point expression $\textbf{fix}\ f.\, f$ which does not have a value. In the previous interpreter, it would just step to itself; here it loops and never invoked the original identity continuation.

```
fail
eval 10000 black_hole = eval_ (fold 'fix (\f. f))
```

We limit here the interpreter to 10000 steps and then fail, which is indicated by the `fail` keyword preceding the definition.

The live code for this interpreter can be found in the file cps.cbv. where "cps" stands for continuation-passing style. It is quite a flexible and powerful technique for writing an interpreter.

Just eyeballing it, we can see it more concise than the one we derived from the stepping and value judgments. It is an interesting exercise to reverse engineering a *judgment* whose computational interpretation reading corresponds to this interpreter. This is the K machine (see, for example, Lecture 12 of the 2021 edition of this course.

It is by no means obvious that the two interpreters coincide in the sense of producing the same observable values. One proof for a fragment of the language can be found in Harper [2016, Chapter 28].

Exercises

Exercise 1 Complete the code from eval.cbv to include

(i) Binary sums $\tau + \sigma$

(ii) The empty type 0

(iii) Products $\tau \times \sigma$

(iv) Lazy pairs $\tau \mathbin{\&} \sigma$ (see Exercise L6.6)

(v) Parametric polymorphism $\forall \alpha.\, \tau$

(vi) Fixed points $\textbf{fix}\ f.\, e$

Exercise 2 Complete the code from cps.cbv to include

(i) Binary sums $\tau + \sigma$

(ii) The empty type 0

(iii) Products $\tau \times \sigma$

(iv) Lazy pairs $\tau \mathbin{\&} \sigma$ (see Exercise L6.6)

(v) Parametric polymorphism $\forall \alpha.\, \tau$

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

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.