

Systematic Compiler Construction

Michael Sperber Peter Thiemann

January 15, 2004

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999; Peter Thiemann, 2000, 2002

Chapter 4

Implementing the Lambda Calculus

The last chapter has provided the necessary machinery to translate Mini-Caml into a program schemes which are merely a straightforward extension to the lambda calculus. The language of program schemes is already considerably simplified compared to the abstract syntax of Mini-Caml. Implementing the lambda calculus means implementing Mini-Caml.

As regular and simple as the lambda calculus is, it is still very different from the machine language of real-world processors. Therefore, implementing the lambda calculus still involves a number of intermediate steps until it is suitable for native code generation. The first step is the derivation of an interpreter from the denotational semantics. From that follows an interpreter written in so-called *continuation-passing style* (or *CPS*) which is still visibly correct but closer to machine language. The CPS interpreter induces yet another intermediate language for programs, and a new sequence of interpreters implements CPS, each one getting closer to machine language.

4.1 Semantics into interpretation

The denotational semantics translates straightforwardly into an interpreter.

The output of the compiler phase described in the last chapter is an object of the type `scheme` in the `Lambda` structure:

```
type const =
  CInt of int
  | CString of string
  | CChar of char

type exp =
  Const of const
  | Ident of string
  | Builtin of string
  | Lambda of string * exp
  | Apply of exp * exp
  | Let of string * exp * exp
  | If of exp * exp * exp

type equation = string * exp
```

```
type scheme = equation list * exp
```

The datatype corresponding to *Val* is a sum type with identical structure except for a few additions:

```
type value =
  Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> value)
  | Cons of value * value
  | Wrong
```

Anticipating the demands of Mini-Caml, the `value` type contains a few more components than *Val*:

- `Unit` is the value of the unit constructor `()` in Caml, and `EmptyList` represents the empty list `[]`.
- `String` and `Char` are constructors for the corresponding constants in Caml.
- `Cons` is a constructor for pairs—aggregate data structures with two arbitrary components. In particular, they serve to implement lists.

Since the Mini-Caml abstract syntax distinguishes built-in identifiers and constants from normal ones, both of which are constants in the view of the denotational semantics. Therefore, the meaning function α has two counterparts in the implementation:

```
let eval_const c =
  match c with
  | CInt i -> Int i
  | CString s -> String s
  | CChar c -> Char c
```

The meaning function for built-in identifiers is `eval_builtin`. The essential first-order constants are trivial to implement:

```
let eval_builtin i =
  match i with
  | "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
```

Function constants must perform the same case analyses and injections as their counterparts in the semantics:

```
| "inc" ->
  Fun
    (function x ->
      match x with
      | Int x' -> Int (x' + 1)
      | _ -> Wrong)
```

```

| "dec" ->
  Fun
    (function x ->
      match x with
      Int x' -> Int (x' - 1)
      | _ -> Wrong)
| "zerop" ->
  Fun
    (function x ->
      match x with
      Int x' -> Bool (x' = 0)
      | _ -> Wrong)
| "+" ->
  Fun
    (function x ->
      Fun
        (function y ->
          match x with
          Int x' ->
            (match y with
              Int y' -> Int (x' + y')
              | _ -> Wrong)
          | _ -> Wrong))

```

Equality also comes in handy:

```

| "=" ->
  Fun (function x -> Fun (function y -> Bool (x = y)))

```

Also, identifiers named as in Caml are responsible for list construction and selection:

```

| "::" ->
  Fun
    (function x ->
      Fun
        (function y ->
          Cons (x, y)))
| "hd" ->
  Fun
    (function x ->
      match x with
      Cons (h, _) -> h
      | _ -> Wrong)
| "tl" ->
  Fun
    (function x ->
      match x with
      Cons (_, t) -> t
      | _ -> Wrong)
| _ -> Wrong

```

This list is necessarily incomplete and arbitrary.

Environments in the denotational semantics are functions, and it is natural to do the same in a functional programming language:

```

let empty_env x = raise Not_found

```

```
let extend_env env x y =
  function z ->
    if z = x
    then y
    else env z
```

Whereas the denotational semantics applies environments directly, the interpreter goes through a function `lookup_env` to do this. This makes it possible to change the representation for environments later into something more efficient without compromising the interpreter.

The semantics for expressions corresponds to a function `eval` which must have the expression to be evaluated, a function environment, and an environment. Again, the code corresponds closely to the semantics. Constants first:

```
let rec eval e fenv env =
  match e with
  Const c -> eval_const c
```

For identifiers, the interpreter must distinguish between locally bound identifiers, function names, and built-ins:

```
| Ident i ->
  (try env i
   with Not_found -> fenv i)
| Builtin b -> eval_builtin b
```

The rest is a one-to-one transliteration of the semantics:

```
| Lambda (x, e) ->
  Fun
  (function y ->
   eval e fenv (extend_env env x y))
| Apply (e1, e2) ->
  (match eval e1 fenv env with
   Fun f -> f (eval e2 fenv env)
  | _ -> Wrong)
| Let (x, e', e) ->
  eval e fenv (extend_env env x (eval e' fenv env))
| If (e, e1, e2) ->
  (match eval e fenv env with
   Bool t ->
   if t
   then eval e1 fenv env
   else eval e2 fenv env
  | _ -> Wrong)
```

The same holds true for the top-level evaluation function:

```
let eval_program (functions, e) =
  let rec fenv f =
    eval (List.assoc f functions) fenv empty_env
  in
  eval e fenv empty_env
```

4.2 Writing a definitional interpreter

This naive interpreter is surprisingly simple, mainly because the language being interpreted and the language the interpreter is written (the metalanguage) in are so similar. The correctness of the interpreter with respect to the denotational semantics seems obvious. However, there are a few crucial issues that require careful consideration before the above interpreter can be deemed correct:

1. Function application in the metalanguage follows the *call-by-value* evaluation strategy. Therefore, the interpreter also follows the call-by-value semantics, even though it is textually derived from the call-by-name semantics.
2. The interpreter handles parameter passing and return of the interpreted language by parameter passing and return of the metalanguage.
3. The interpreter handles binding by depending on the binding policy being the same in interpreted language and metalanguage. Would Caml use dynamic binding, so would the interpreted language.
4. The interpreter handles recursion with `let rec` rather than using an explicit fixpoint combinator. Hence, it uses the “implicit” fixpoint operator of the metalanguage.

Consequently, even though the naive interpreter seems “obviously” correct, it makes use of quite a few more or less arbitrary properties of the metalanguage. Were the metalanguage to change its semantics, the language semantics would change as well. The interpreter is—on its own—not quite *definitional* yet as it delegates important semantics issues to the metalanguage rather than taking a stand on its own.

Of all the above issues, the second is the most important: Assembler programmers know that function application involves some fairly complicated machinery: it may store a “return address,” create “stack frames,” advance a “stack pointer” and jumps to a “target address.” Thus, the interpreter shown above really fails to reveal a very important mechanism crucial to code generation at a later stage. As it turns out, resolving the procedure application issue also solves the first issue of pinning down an evaluation strategy. The binding policy as well as parameter passing will turn out to be easy to handle and is therefore left to a later section. The final issue of recursion is fortunately of little relevance as all of our intermediate languages—including machine code—possess an implicit recursion mechanism which a compiler can simply use.

The trick in making the interpreter definitional with respect to procedure application and return is to have it use a more primitive notion instead of the corresponding metalanguage constructs. Thus, the new interpreter will make two restrictive assumptions:

- The metalanguage allows only function applications that are tail calls.
- The metalanguage does not allow values to be returned from function applications.

These restrictions turn procedure applications into jumps with parameter passing. Both are mechanisms that translate straightforwardly into machine code.

The crucial idea in rewriting the interpreter to follow these restrictions is to recognize what actually happens with a value after the interpreter has computed it: That value is “passed” to some surrounding computation derived from its context. Look at the following expression

$$(f\ 2\ (g\ 23\ 17))$$

The interpreter computes—according to the rules of call-by-value evaluation—the value of $(g\ 23\ 17)$ first, and then passes it to a computation corresponding to the evaluation context $(f\ 2\ [])$. This context really represents the future of the computation of $(g\ 23\ 17)$. In the naive interpreter, this future is always implicit somewhere in the folds of the implementation of the metalanguage. For effective compilation, it is necessary to make this context computation visible. It is called a *continuation*. In the terminology of the denotational semantics, it is a function $k : Val \rightarrow Val$, in this case $k = \lambda x.f\ 2\ x$.

The new interpreter will, instead of returning the value of an expression (a facility that is no longer available), pass it to a continuation corresponding to the context of that expression. This continuation (the *current continuation*) is an additional parameter to the new `eval` function. For the transition to happen, it is necessary to modify the `value` data type slightly: Since a function embedded in `value` will also no longer be able to return directly, it must take a continuation argument:

```
type value =
  Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> continuation -> value)
  | Cons of value * value
  | Wrong
and continuation = value -> value
```

Constants are evaluated as before:

```
let eval_const c =
  match c with
  | CInt i -> Int i
  | CString s -> String s
  | CChar c -> Char c
```

The new `eval_builtin` function handles first-order constants the same as before:

```
let eval_builtin i =
  match i with
  | "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
```

For function built-ins, the injected functions also need to accept a continuation argument and pass their results to it. The `inc` function is a unary example:

```
| "inc" ->
  Fun
    (function x -> function k ->
      k
        (match x with
         | Int x' -> Int (x' + 1)
         | _ -> Wrong))
```

The whole business is somewhat more tedious for the binary case:

```

| "+" ->
  Fun
  (function x -> function k ->
    k (Fun
      (function y -> function k ->
        k
          (match x with
            Int x' ->
              (match y with
                Int y' -> Int (x' + y')
                | _ -> Wrong)
            | _ -> Wrong))))

```

Environments are as before.

The new `eval` function takes a continuation argument:

```

let rec eval e fenv env k =
  match e with

```

The interpreter passes constants to the current continuation:

```

  Const c -> k (eval_const c)

```

Identifiers are handled the same as before:

```

| Ident i ->
  k
  (try env i
    with Not_found -> fenv i)
| Builtin b -> k (eval_builtin b)

```

The abstractions embedded into `value` need to accept a continuation argument:

```

| Lambda (x, e) ->
  k (Fun
    (function y -> function k ->
      eval e fenv (extend_env env x y) k))

```

Correspondingly, application needs to provide suitable continuations so that it can process the values further:

```

| Apply (e1, e2) ->
  eval
  e1 fenv env
  (function e1_val ->
    eval
    e2 fenv env
    (function e2_val ->
      match e1_val with
        Fun f -> f e2_val k
        | _ -> k Wrong))

```

The same holds true for `Let`:

```

| Let (x, e', e) ->
  eval
  e' fenv env
  (function e'_val ->
    eval e fenv (extend_env env x e'_val) k)

```

... as well as for the conditional:

```
| If (e, e1, e2) ->
  eval
    e fenv env
  (function e_val ->
    match e_val with
      Bool t ->
        if t
          then eval e1 fenv env k
          else eval e2 fenv env k
      | _ -> k Wrong)
```

Ultimately, the final return value of the program needs to be passed to a trivial identity continuation:

```
let eval_program (functions, e) =
  let rec fenv f =
    eval (List.assoc f functions) fenv empty_env (function v -> v)
  in
  eval e fenv empty_env (function v -> v)
```

The new interpreter uses a programming technique or style known as *continuation-passing style* or *CPS*. CPS is an important tool both in denotational semantics and programming pragmatics. It helps in this case because the CPS interpreter has the following properties:

1. The only non-tail calls in the program are either to metalanguage primitives such as `Int` or `match` or to functions like `eval_const` or `eval_builtin` which themselves contain no calls to other functions and thus could also be expanded into `eval`. None of them requires sophisticated machinery, as they call no other functions, and thus will typically be compiled “in-line” without involving function calls. The only function which ever “does” something on return is the inevitable identity continuation passed at top level. Of course, that continuation represents the empty context, and thus does nothing.
2. The code is suddenly “linear”: Computations actually occur in the order in which they actually happen at interpretation time. In fact, this pins down the evaluation strategy: Even if the metalanguage were to use call-by-name, the CPS interpreter would still implement call-by-value semantics.
3. Every intermediate result has a name—the name of the continuation argument.

All these properties bring the CPS interpreter considerably closer to machine language than the naive version, and is also quite definitional already. Remember that machine language also has the following properties:

- Machine languages only have tail calls in the form of jumps. (Admittedly, CISCs usually have non-tail calls in the form of subroutine calls as well, but RISCs often do not.)
- Machine language is linear.
- Every intermediate result has a name, either the name of a register or of a storage location.

Still, the final objective of compilation is, of course, a compiler that *translates to* machine language rather than an interpreter *written in* machine language. In the compiler, therefore, it is necessary to translate our input program into CPS to make use of its useful properties.

4.3 Non-local exits

The CPS transformation has yet another pleasant side-effect: So far we have ignored the `raise` and `try` constructs of Mini-Caml in our translation of the lambda calculus. The pragmatic view to implementing this is to see `raise` and `try` as primitives. Since `try` is a special piece of it is necessary to transform

```
try e x -> e'
```

into

```
try (function () -> e) (function x -> e')
```

and have `try` apply the thunk that represents e . With the direct-style interpreter, the only way of implementing `try` and `raise` is by using the corresponding constructs in the metalanguage. This, however, explains nothing, especially since the denotational semantics of the lambda calculus contains no provision for exceptions. In a definitional interpreter, therefore, this solution is not acceptable.

What does `try` do? It installs an *exception handler* which a subsequent `raise` invokes. Moreover, `raise` continues execution with the `try` construct. In other words: the continuation of `raise` is the continuation of the handler expression in the corresponding `try`. To properly handle this, the interpreter needs to propagate an additional continuation in addition to the current one: the *exception handler continuation*. The value domain needs to be extended once more:

```
type value =
  Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Cons of value * value
  | Wrong
and continuation = value -> value
and handler = value -> value
```

The evaluation functions which previously accepted a current continuation argument now also accept a handler argument `h`:

```
let eval_builtin i =
  match i with
```

Naturally, all created functions must be extended. For example:

```
| "inc" ->
  Fun
    (function x -> function h -> function k ->
      k
        (match x with
          Int x' -> Int (x' + 1)
          | _ -> Wrong))
```

Analogous changes propagate through the central interpreter. These are all straightforward, however: The interesting cases are, of course, the primitive definitions for `try` and `raise` themselves:

```

| "try" ->
  Fun
  (function thunk -> function h -> function k ->
    k (Fun
      (function handler -> function h -> function k ->
        match thunk with
          Fun thunk' ->
            (match handler with
              Fun handler' ->
                thunk' Unit
                (function exc ->
                  handler' exc h k)
                k
                | _ -> k Wrong)
              | _ -> k Wrong)))

```

Note how the current continuation of the handler becomes the current continuation of the try application. The raise primitive just calls the handler, discarding the current continuation:

```

| "raise" ->
  Fun
  (function exc -> function h -> function k ->
    h exc)

```

The eval and eval_program functions essentially stay as before except for the additional current handler h being passed around:

```

let rec eval e fenv env h k =
  match e with
  | Const c -> k (eval_const c)
  | Ident i ->
    k
    (try env i
      with Not_found -> fenv i)
  | Builtin b -> k (eval_builtin b)
  | Lambda (x, e) ->
    k (Fun
      (function y -> function h -> function k ->
        eval e fenv (extend_env env x y) h k))
  | Apply (e1, e2) ->
    eval
    e1 fenv env
    h
    (function e1_val ->
      eval
      e2 fenv env
      h
      (function e2_val ->
        match e1_val with
          Fun f -> f e2_val h k
          | _ -> k Wrong))
  | Let (x, e', e) ->
    eval
    e' fenv env

```

```

      h
      (function e'_val ->
        eval e fenv (extend_env env x e'_val) h k)
| If (e, e1, e2) ->
  eval
  e fenv env
  h
  (function e_val ->
    match e_val with
    Bool t ->
      if t
      then eval e1 fenv env h k
      else eval e2 fenv env h k
    | _ -> k Wrong)

exception Not_handled

let eval_program (functions, e) =
  let rec fenv f =
    eval
      (List.assoc f functions) fenv empty_env
      (function _ -> raise Not_handled)
      (function v -> v)
  in
  eval
    e fenv empty_env
    (function _ -> raise Not_handled)
    (function v -> v)

```

4.4 The CPS Transformation

Continuations have become so important in compiler construction that CPS warrants a closer look and more systematic study. Indeed, understanding CPS is a prerequisite to many newer research papers in compiler construction. At the heart of CPS is the *CPS transformation* which transforms a term in the lambda calculus into one using explicit continuations. This new representation for lambda terms (and, hence, the transformation) is exactly what is needed to reap the benefits of CPS as described in the previous section.

The presentation here follows the work by Danvy and Filinski [DF92].

For expository purposes, we will revert to the pure lambda calculus. All results will carry over smoothly to its applied variants. However, the notation of application (formerly $e_1 e_2$) will change to $@_{e_1} e_2$.

4.4.1 Classical CPS transformation

The central idea of the classical CPS transformation is by Plotkin and Fischer [Fis93, Plo75]. Recall that the interpreter used abstractions to represent continuations. The CPS transformation does the same.

4.1 Definition (Fischer/Plotkin Call-by-Value CPS Transformation)

It is a function $\llbracket _ \rrbracket : E \rightarrow E$:

$$\begin{aligned} \llbracket x \rrbracket &:= \lambda k. @ k x \\ \llbracket \lambda x. e \rrbracket &:= \lambda k. @ k (\lambda x. \llbracket e \rrbracket) \\ \llbracket @ e_1 e_2 \rrbracket &:= \lambda k. @ \llbracket e_1 \rrbracket (\lambda v_1. @ \llbracket e_2 \rrbracket (\lambda v_2. @ (@ v_1 v_2) k)) \quad (v_1, v_2 \text{ fresh}) \end{aligned}$$

□

The Fischer/Plotkin CPS transformation is simple enough, and the following statement states its correctness with respect to call-by-value evaluation:

4.2 Theorem (Simulation)

Let e be a lambda term. Let furthermore eval_v be call-by-value evaluation.

$$\text{eval}_v(e) = \text{eval}_v(@ \llbracket e \rrbracket (\lambda x. x))$$

□

Moreover, the CPS transformation has a pleasant side effect:

4.3 Theorem (Indifference)

Let e be a lambda term. Let furthermore eval_v be call-by-value evaluation and eval_n be call-by-name evaluation.

$$\text{eval}_n(@ \llbracket e \rrbracket (\lambda x. x)) = \text{eval}_v(@ \llbracket e \rrbracket (\lambda x. x))$$

□

The consequence of the indifference property is that the CPS-transformed term is indifferent to the evaluation strategy: Call-by-name and call-by-value will produce the same result on a CPS term. Consequently, the CPS interpreter is now independent of the evaluation strategy of the metalanguage, which brings it a significant step closer to being definitional.

4.4.2 Avoiding administrative β redexes

Unfortunately, the Fischer/Plotkin CPS transformation is not suitable for direct application in realistic compilers: it produces humungous result terms. For example, the CPS version of $@(\lambda x. x)(@y y)$ is this:

$$\begin{aligned} &\lambda k. @(\lambda k. @ k (\lambda x. \lambda k. @ k x)) \\ &(\lambda m. @(\lambda k. @(\lambda k. @ k y)(\lambda m. @(\lambda k. @ k y)(\lambda n. @(@m n) k)))(\lambda n. @(@m n) k)) \end{aligned}$$

This term contains a large number of β redexes—in addition to the *beta* redex already present in the original term. Reducing those *administrative redexes* leads to the following, much more acceptable term:

$$\lambda k. @ (y y) (\lambda a. @ (@ (\lambda x. \lambda k. (@ k x)) a) (\lambda a. @ k a))$$

Hence, for practical intents and purposes, the Fischer/Plotkin CPS transformation needs to be accompanied by a post-reducer which removes the β reductions introduced by the vanilla transformation. This approach has the disadvantage that it still constructs the intermediate, large CPS term only to replace it immediately by something much smaller. It is much more desirable to compute the final result directly without large intermediate terms.

The method to achieve this “on-the-fly” post-reduction is to classify the abstractions and applications on the right-hand sides of the transformation into those

which will be part of an administrative β redex and those which will not. With the straightforward Fischer/Plotkin transformation, this is not possible—some abstractions and applications sometimes do take part in administrative redexes, and sometimes do not. However, it is possible to perform η expansion on the right-hand sides in a few, select instances, and then perform the classification.

The new transformation resulting from this has annotations on each λ and each $@$ indicating its classification: $\bar{\lambda}$ is for *static* abstractions that are part of administrative redexes (and therefore do not show up in the result term), and $\underline{\lambda}$ is for *dynamic* abstractions which definitely are part of the transformed term. Analogously, $\underline{@}$ denotes a dynamic application, and $\bar{@}$ a static one. The reformulation of the transformation is due to Danvy and Filinski [DF92], hence:

4.4 Definition (Danvy/Filinski CPS Transformation)

Let e be a lambda term. The Danvy/Filinski CPS Transformation is a function $\llbracket _ \rrbracket : E \rightarrow (E \rightarrow E) \rightarrow E$:

$$\begin{aligned} \llbracket x \rrbracket &:= \bar{\lambda}\kappa.\bar{@}\kappa x \\ \llbracket \lambda x.e \rrbracket &:= \bar{\lambda}\kappa.\bar{@}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\bar{@}\llbracket e \rrbracket)(\bar{\lambda}v.\underline{@}k v)) \\ \llbracket @e_1 e_2 \rrbracket &:= \bar{\lambda}\kappa.\bar{@}\llbracket e_1 \rrbracket (\bar{\lambda}v_1.\bar{@}\llbracket e_2 \rrbracket (\bar{\lambda}v_2.\underline{@}(\underline{@}v_1 v_2) (\underline{\lambda}a.\bar{@}\kappa a))) \end{aligned}$$

□

Note that, in this definition, κ stands for a continuation *at transformation time*; only k ever appears in the output of the transformation

The corresponding correctness statement is this:

4.5 Theorem

For a lambda term e , $\underline{\lambda}k.\bar{@}\llbracket e \rrbracket(\bar{\lambda}v.\underline{@}k v)$ is $\beta\eta$ -equivalent to the corresponding result term of the Fischer/Plotkin transformation.

□

The implementation of the Danvy/Filinski transformation is straightforward: Static abstractions and applications become the corresponding constructs in the metalanguage, and their dynamic counterparts become syntax constructors.

The introduction of additional η redexes allows for the classification of the applications and abstractions of a lambda term. Mostly, they participate in administrative redexes and therefore do not show up in the resulting term. However, there is an exception:

$$\bar{@}\llbracket \lambda f.@f x \rrbracket(\bar{\lambda}v.v) = \lambda f.\lambda k.(@f x) (\lambda a.@k a)$$

The residual term still contains an η redex introduced by the CPS transformation. This has potentially serious consequences, as the application in the original term is a tail call. Some modern programming languages based on the lambda calculus (most notably Scheme) demand that tail calls do not create new continuations. However, the above η redex is just that.

Therefore, it is necessary to augment the transformation to guard against this case. The idea is to duplicate the transformation rules: A new version of the rules is for “trivial” continuations of the form $\bar{\lambda}v.\underline{@}k v$; the new rules avoid building the redex in the residual rules. The copied rules are called $\llbracket _ \rrbracket'$ in the following definition.

4.6 Definition (Tail-Recursive Danvy/Filinski CPS Transformation)

Let e be a lambda term. The tail-recursive Danvy/Filinski CPS Transformation is

a function $\llbracket _ \rrbracket : E \rightarrow (E \rightarrow E) \rightarrow E$:

$$\begin{aligned} \llbracket x \rrbracket &:= \bar{\lambda}\kappa.\bar{\textcircled{a}}\kappa x \\ \llbracket \lambda x.e \rrbracket &:= \bar{\lambda}\kappa.\bar{\textcircled{a}}\kappa(\underline{\lambda}x.\underline{\lambda}k.(\bar{\textcircled{a}}\llbracket e \rrbracket'k)) \\ \llbracket \textcircled{a}e_1 e_2 \rrbracket &:= \bar{\lambda}\kappa.\bar{\textcircled{a}}\llbracket e_1 \rrbracket (\bar{\lambda}v_1.\bar{\textcircled{a}}\llbracket e_2 \rrbracket) (\bar{\lambda}v_2.\underline{\textcircled{a}}(\underline{\textcircled{a}}v_1 v_2) (\underline{\lambda}a.\bar{\textcircled{a}}\kappa a)) \end{aligned}$$

where $\llbracket _ \rrbracket' : E \rightarrow E \rightarrow E$:

$$\begin{aligned} \llbracket x \rrbracket' &:= \bar{\lambda}k.\underline{\textcircled{a}}k x \\ \llbracket \lambda x.e \rrbracket' &:= \bar{\lambda}k.\underline{\textcircled{a}}k(\underline{\lambda}x.\underline{\lambda}k.(\bar{\textcircled{a}}\llbracket e \rrbracket'k)) \\ \llbracket \textcircled{a}e_1 e_2 \rrbracket' &:= \bar{\lambda}k.\bar{\textcircled{a}}\llbracket e_1 \rrbracket (\bar{\lambda}v_1.\bar{\textcircled{a}}\llbracket e_2 \rrbracket) (\bar{\lambda}v_2.\underline{\textcircled{a}}(\underline{\textcircled{a}}v_1 v_2) k) \end{aligned}$$

□

Note that Theorem 4.5 requires a slight reformulation: The result of transforming a term e into CPS in a dynamic context is given by $\underline{\lambda}k.\bar{\textcircled{a}}\llbracket e \rrbracket'k$.

4.5 Implementing the CPS Transformation

For implementing the CPS transformation, it is necessary to look at the form of the lambda terms generated by the transformation. It turns out that the transformation creates only a subset of all lambda terms; this subset lends itself to a specialized representation which makes implementing the transformation easier:

4.7 Theorem (Language of CPS terms)

The output of the tail-recursive Danvy/Filinski CPS transformation, $\underline{\lambda}k.\bar{\textcircled{a}}\llbracket E \rrbracket'k$, is exactly the language of the following grammar with start symbol $\langle \text{top-exp} \rangle$.

$$\begin{aligned} \langle \text{val-exp} \rangle &::= \langle \text{const} \rangle \mid \langle \text{var} \rangle \mid \lambda \langle \text{var} \rangle . \lambda \mathbf{k} . \langle \text{exp} \rangle \\ \langle \text{exp} \rangle &::= \textcircled{a} \mathbf{k} \langle \text{val-exp} \rangle \\ &\mid \text{let } \langle \text{var} \rangle = \langle \text{val-exp} \rangle \text{ in } \langle \text{exp} \rangle \\ &\mid \text{if } \langle \text{val-exp} \rangle \text{ then } \langle \text{exp} \rangle \text{ else } \langle \text{exp} \rangle \\ &\mid \textcircled{a} (\textcircled{a} \langle \text{val-exp} \rangle \langle \text{val-exp} \rangle) (\lambda \langle \text{var} \rangle . \langle \text{exp} \rangle) \\ &\mid \textcircled{a} (\textcircled{a} \langle \text{val-exp} \rangle \langle \text{val-exp} \rangle) \mathbf{k} \\ \langle \text{top-exp} \rangle &::= \lambda \mathbf{k} . \langle \text{exp} \rangle \end{aligned}$$

□

Note that Theorem 4.7 is in light of a CPS transformation with the following rules for the applied lambda calculus:

$$\begin{aligned} \llbracket \text{let } x = e' \text{ in } e \rrbracket &:= \bar{\lambda}\kappa.\bar{\textcircled{a}}\llbracket e' \rrbracket (\bar{\lambda}v_1.\underline{\text{let}} x' = v_1 \underline{\text{in}} \bar{\textcircled{a}}\llbracket e[x \rightarrow x'] \rrbracket \kappa) \\ \llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket &:= \bar{\lambda}\kappa.\bar{\textcircled{a}}\llbracket e \rrbracket (\bar{\lambda}b.\underline{\text{if}} b \underline{\text{then}} \bar{\textcircled{a}}\llbracket e_1 \rrbracket \kappa \underline{\text{else}} \bar{\textcircled{a}}\llbracket e_2 \rrbracket \kappa) \end{aligned}$$

This straightforward rule for *if* unfortunately has a pragmatic problem: The transformation propagates the context of an *if* term into both its branches, thereby duplicating it. For example, the lambda term

$$\textcircled{a}f (\text{if } (\text{if } x \text{ then } y \text{ else } z) \text{ then } 4 \text{ else } 5)$$

results in a rather large CPS term:

$$\begin{aligned} \underline{\lambda}k.\text{if } x \text{ then } (\text{if } y \text{ then } \textcircled{a}(\textcircled{a}f 4) (\underline{\lambda}v.\underline{\textcircled{a}}k v) \text{ else } \textcircled{a}(\textcircled{a}f 4) (\underline{\lambda}v.\underline{\textcircled{a}}k v)) \\ \text{else } (\text{if } z \text{ then } \textcircled{a}(\textcircled{a}f 4) (\underline{\lambda}v.\underline{\textcircled{a}}k v) \text{ else } \textcircled{a}(\textcircled{a}f 4) (\underline{\lambda}v.\underline{\textcircled{a}}k v)) \end{aligned}$$

This excessive code duplication is undesirable in realistic compiler. It may therefore be preferable to cut off the context before moving onto the branches of an *if* expression:

$$\llbracket \text{if } e \text{ then } e_1 \text{ else } e_2 \rrbracket := \overline{\lambda} \kappa. \underline{\text{let}} \ k = \lambda a. \overline{\text{@}} \kappa \ a \\ \underline{\text{in}} \ \overline{\text{@}} \llbracket e \rrbracket \ (\overline{\lambda} b. \underline{\text{if}} \ b \ \text{then} \ \overline{\text{@}} \llbracket e_1 \rrbracket' \ k \ \text{else} \ \overline{\text{@}} \llbracket e_2 \rrbracket' \ k)$$

With this rule in place, the CPS transformation is again “linear”: The size of the output terms is linear in the size of the input terms. The new translation also introduces a new rule into the grammar for CPS terms:

$$\langle \text{exp} \rangle ::= \text{let } k = (\lambda \langle \text{var} \rangle. \langle \text{exp} \rangle) \ \text{in} \ \langle \text{exp} \rangle$$

Now the machinery is in place to actually implement the transformation. The most straightforward method would be to translate `Lambda.exp` terms into `Lambda.exp` terms. However, this loses precious information from the CPS transformation: The residual terms do not distinguish between continuations and ordinary values, and it is difficult to special-case on the subset of `Lambda.exp` terms that the transformation produces. Theorem 4.7 provides the basis for a more specialized representation of CPS terms which goes into the interface of a structure named `Cps`:

```
type ident = string

type valexp =
  Const of Lambda.const
  | Ident of ident
  | Builtin of ident
  | Lambda of ident * ident * exp
and exp =
  Return of ident * valexp
  | Let of ident * valexp * exp
  | If of valexp * exp * exp
  | Call of valexp * valexp * cont
  | TailCall of valexp * valexp * ident
  | LetCont of ident * cont * exp
and cont = ident * exp
and cps = WithCont of ident * exp
and program = (ident * cps) list * cps
```

The transformation requires two identifiers, a unique constant one for continuations that corresponds to the variable *k* in the formal description:

```
let ki = Rename.generate_unique "k"
... and a seed for fresh “normal” identifiers:
let ai = "a"
```

The implementation of the transformation proper very closely follows the Danvy/Filinski formulation. First, the counterpart for $\llbracket _ \rrbracket$:

```
let rec from_lambda_1 e k =
  match e with
  | Lambda.Const c -> k (Const c)
  | Lambda.Ident i -> k (Ident i)
  | Lambda.Builtin b -> k (Builtin b)
```

```

| Lambda.Lambda (x, e) ->
  k (Lambda (x, ki, from_lambda_2 e ki))
| Lambda.Apply(e1, e2) ->
  from_lambda_1 e1
  (function v1 ->
    from_lambda_1 e2
    (function v2 ->
      let a = Rename.generate_unique ai in
      Call (v1, v2, (a, k (Ident a))))))
| Lambda.Let(x, e', e) ->
  from_lambda_1 e'
  (function v' ->
    Let(x, v', from_lambda_1 e k))
| Lambda.If(e, e1, e2) ->
  let a = Rename.generate_unique ai in
  LetCont(ki, (a, k (Ident a)),
    from_lambda_1 e
    (function v ->
      If(v, from_lambda_2 e1 ki, from_lambda_2 e2 ki)))

```

Now $\llbracket _ \rrbracket'$:

```

and from_lambda_2 e ki =
  match e with
  | Lambda.Const c -> Return (ki, Const c)
| Lambda.Ident i -> Return (ki, Ident i)
| Lambda.Builtin b -> Return (ki, Builtin b)
| Lambda.Lambda (x, e) ->
  Return (ki, (Lambda (x, ki, from_lambda_2 e ki)))
| Lambda.Apply(e1, e2) ->
  from_lambda_1 e1
  (function v1 ->
    from_lambda_1 e2
    (function v2 ->
      TailCall (v1, v2, ki)))
| Lambda.Let(x, e', e) ->
  from_lambda_1 e'
  (function v' ->
    Let(x, v', from_lambda_2 e ki))
| Lambda.If(e, e1, e2) ->
  from_lambda_1 e
  (function v ->
    If(v, from_lambda_2 e1 ki, from_lambda_2 e2 ki))

```

Two trivial functions take care of top-level expressions and program schemes:

```

let from_lambda_top e = WithCont(ki, from_lambda_2 e ki)

let from_lambda (equations, body) =
  (List.map
    (function (name, e) -> (name, from_lambda_top e))
    equations,
  from_lambda_top body)

```

4.6 Implementing CPS

Of course, the new representation requires a new interpreter formulated by specializing the original CPS interpreter for the standard lambda calculus. Thus, it reverts to explicitly passing around exception handlers and continuations. The value domain adds a `Cont` constructor:

```
type value =
  Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Cont of continuation
  | Cons of value * value
  | Wrong
and continuation = value -> value
and handler = value -> value
```

The interpreter starts with constants:

```
let eval_const c =
  match c with
  | Lambda.CInt i -> Int i
  | Lambda.CString s -> String s
  | Lambda.CChar c -> Char c
```

The `eval_builtin` function is the same as the one in the CPS interpreter for the ordinary lambda calculus as well as the functions for handling environments. The `eval` function is responsible for `Cps.expr` terms:

```
let rec eval e fenv (env : Cps.ident -> value) h =
  match e with
  | Return (ki, ve) ->
    let v = eval_val ve fenv env in
    let Cont k = env ki in
    k v
  | Let (x, ve, e) ->
    eval e fenv (extend_env env x (eval_val ve fenv env)) h
  | If (ve, e1, e2) ->
    (match eval_val ve fenv env with
     | Bool t ->
       if t
       then eval e1 fenv env h
       else eval e2 fenv env h)
  | Call(ve1, ve2, c) ->
    let v1 = eval_val ve1 fenv env in
    let v2 = eval_val ve2 fenv env in
    let Cont k = eval_cont c fenv env h in
    (match v1 with
     | Fun f -> f v2 h k
     | _ -> k Wrong)
  | TailCall(ve1, ve2, ki) ->
    let v1 = eval_val ve1 fenv env in
```

```

let v2 = eval_val ve2 fenv env in
let Cont k = env ki in
(match v1 with
  Fun f -> f v2 h k
  | _ -> k Wrong)
| LetCont(ki, c, e) ->
  eval e fenv
  (extend_env env ki (eval_cont c fenv env h))
  h

```

The `eval_val` function handles terms of type `Cps.valexp`:

```

and eval_val ve fenv env =
  match ve with
  Const c -> eval_const c
  | Ident i ->
    (try env i
     with Not_found -> fenv i)
  | Builtin b -> eval_builtin b
  | Lambda(x, ki, e) ->
    Fun
    (function y -> function h -> function k ->
     eval e fenv
     (extend_env (extend_env env x y) ki (Cont k))
     h)

```

The `eval_cont` function handles terms of type `Cps.cont`:

```

and eval_cont (x, e) fenv env h =
  Cont
  (function v ->
   eval e fenv (extend_env env x v) h)

```

Finally, `eval_top` handles top-level expressions:

```

let eval_top t fenv =
  match t with
  WithCont (ki, e) ->
    eval e fenv
    (extend_env empty_env ki (Cont (function v -> v)))
    (function _ -> raise Not_found)

```

The new `eval_program` evaluates the right-hand sides of the program scheme equations prior to interpreting the body, to more closely model the semantics of Caml:

```

let eval_program (equations, body) =
  let rec fenv f =
    eval_top (List.assoc f equations) fenv
  in
  eval_top body fenv

```

4.7 Making Functions Machine-Friendly

The next candidates for a representation change are the `Fun` values. The previous interpreters have all used the implementation of lexical scoping in the metalanguage to implement lexical scoping in the object language. Consequently, in order to make the interpreter more definitional, the `function` must go. Another look at the old `eval` clause for `Lambda` makes clear what the issues are:

```

| Lambda(x, ki, e) ->
  Fun
    (function y -> function h -> function k ->
      eval e fenv
        (extend_env (extend_env env x y) ki (Cont k))
      h)

```

The value that `eval` returns for `Lambda` contains enough information to contain the application later on, or, more specifically, to evaluate the inner expression.

Obviously, evaluation of the inner expression requires values for `env` (the environment that was current at the time of creation of the `Fun` object), `x` (the identifier of the `Lambda` expression), `ki` (the identifier of the current continuation) `y` (the value that evaluation of `Apply` later passes), and the expression to be evaluated, `e`. Fortunately, `fenv` never changes, so it is not part of the object. The value for `y` only becomes known at application time, so `env`, `x`, `ki`, and `e` remain. An object containing the necessary information to perform an application is called a *closure*. Therefore, `value` receives a new constructor:

```

type value =
  Unit
  | EmptyList
  | Int of int
  | Bool of bool
  | String of string
  | Char of char
  | Fun of (value -> handler -> continuation -> value)
  | Closure of closure
  | Cont of continuation
  | Cons of value * value
  | Wrong
and closure = environment * ident * ident * exp
and environment = ident -> value

```

Functions are not the only values represented by meta-level functions: The same holds true for continuations and exception handlers. Witness `eval_cont` from the previous interpreter:

```

and eval_cont (x, e) fenv env h =
  Cont
    (function v ->
      eval e fenv (extend_env env x v) h)

```

Again, if continuations are to be packaged into non-function values, it is important to look at the values required by the body of the function representing the continuation: the environment `env`, the name of the intermediate result `x`, the exception handler, `h`, and the body of the continuation, `e`. This results in the following additional clause to the type definitions:

```

and continuation =
  Continuation of environment * handler * ident * exp
  | Stop

```

`Stop` is for representing the initial continuation `function v -> v` from the previous interpreter.

Similar thinking results in an analogous datatype for exception handlers:

```
and handler =
  Handler of environment * handler * ident * exp
| Error
```

The `Error` handler is again for representing the initial handler, simply `raise Not_found` in the previous interpreter.

The `value` type still contains the old `Fun` constructor. The interpreter uses `Fun` for primitives, because most of do not have representations as `Lambda` expressions.

The `eval` function starts off similarly to `eval` in the previous interpreter:

```
and eval e fenv env h =
  match e with
  | Return (ki, ve) ->
    let v = eval_val ve fenv env in
    let Cont k = env ki in
    return k v fenv
  | Let (x, ve, e) ->
    eval e fenv (extend_env env x (eval_val ve fenv env)) h
  | If (ve, e1, e2) ->
    (match eval_val ve fenv env with
     | Bool t ->
       if t
       then eval e1 fenv env h
       else eval e2 fenv env h)
```

The only difference is that the current-continuation parameter, `k`, is no longer a function—`eval` cannot call it directly but rather lets an auxiliary function called `return` handle this:

```
and return k v fenv =
  match k with
  | Stop -> v
  | Continuation (env, h, x, e) ->
    eval e fenv (extend_env env x v) h
```

The new interpreter must handle `Closure` values in `Call` and `TailCall` expressions. The code that performs the procedure call is very similar to the code previously inside the function created for `Lambda` abstractions:

```
| Call(ve1, ve2, c) ->
  let v1 = eval_val ve1 fenv env in
  let v2 = eval_val ve2 fenv env in
  let Cont k = eval_cont c fenv env h in
  (match v1 with
   | Fun f -> f v2 h k
   | Closure (closure_env, x, ki, e) ->
     eval e fenv
       (extend_env (extend_env closure_env x v2) ki (Cont k))
     h
   | _ -> return k Wrong fenv)
| TailCall(ve1, ve2, ki) ->
  let v1 = eval_val ve1 fenv env in
  let v2 = eval_val ve2 fenv env in
  let Cont k = env ki in
  (match v1 with
   | Fun f -> f v2 h k
```

```

    | Closure (closure_env, x, ki, e) ->
      eval e fenv
      (extend_env (extend_env closure_env x v2) ki (Cont k))
      h
    | _ -> return k Wrong fenv)

```

The final clause, `LetCont`, is again as before:

```

| LetCont(ki, c, e) ->
  eval e fenv
  (extend_env env ki (eval_cont c fenv env h))
  h

```

The `eval_val` function is also as before except for the `Lambda` case which gets simpler; all the work is now done in `eval` in the `Call` and `TailCall` clauses:

```

and eval_val ve fenv env =
  match ve with
  | Const c -> eval_const c
  | Ident i ->
    (try env i
     with Not_found -> fenv i)
  | Builtin b -> eval_builtin b fenv
  | Lambda(x, ki, e) ->
    Closure (env, x, ki, e)

```

The same holds true for `eval_cont` whose work is mostly done by `return`:

```

and eval_cont (x, e) fenv env h =
  Cont (Continuation (env, h, x, e))

```

Now that continuations are no longer functions, it is necessary to change `eval_builtin` so it also calls `return` instead of `k` directly. Moreover, some primitives need to call `eval` which means that they need to supply an `fenv` parameter—hence, `fenv` must become an additional parameter for `eval_builtin`. Here are some of the simpler cases of `eval_builtin`:

```

let rec eval_builtin i fenv =
  match i with
  | "()" -> Unit
  | "[]" -> EmptyList
  | "true" -> Bool true
  | "false" -> Bool false
  | "inc" ->
    Fun
      (function x -> function h -> function k ->
        return k
        (match x with
         | Int x' -> Int (x' + 1)
         | _ -> Wrong)
        fenv)
  | "+" ->
    Fun
      (function x -> function h -> function k ->
        return k
        (Fun
         (function y -> function h -> function k ->

```

```

    return k
      (match x with
        Int x' ->
          (match y with
            Int y' -> Int (x' + y')
            | _ -> Wrong)
        | _ -> Wrong)
      fenv))
  fenv)

```

Things become interesting with the implementations of `try` and `raise`. Again, some work shifts from `try` to `raise`:

```

| "try" ->
  Fun
    (function thunk -> function h -> function k ->
      return k
        (Fun
          (function handler -> function h -> function k ->
            match thunk with
              Closure (env, x, ki, e) ->
                (match handler with
                  Closure (handler_env, handler_x, handler_ki, handler_e) ->
                    eval e fenv
                      (extend_env (extend_env env x Unit) ki (Cont k))
                      (Handler (extend_env handler_env handler_ki (Cont k),
                                h,
                                handler_x, handler_e))
                  | _ -> return k Wrong fenv)
                | _ -> return k Wrong fenv))
            fenv)
        | "raise" ->
          Fun
            (function exc -> function h -> function k ->
              match h with
                Handler (env, h, x, e) ->
                  eval e fenv (extend_env env x exc) h
                | Error -> raise Not_found)
            )

```

The representation of the local environment in our interpreter as a function is close to its original definition in the semantics, but unsuitable for implementation on a real machine. A better representation is in order which will come first.

Environments become a pair consisting of two parallel lists: One containing identifiers, one containing the values bound to them.

```

type ident = string
type ident environment = ident list * value list
let empty_env = ([], [])

let extend_env env x y =
  let (identifiers, values) = env in
  (x :: identifiers, y :: values)

let find_position x l =
  let rec loop l pos =

```

```

    match l with
      y::ys -> if x = y then Some pos else loop ys (pos + 1)
    | [] -> None
  in loop l 0

let lookup_env env x =
  let (identifiers, values) = env in
  match find_position x identifiers with
    Some pos -> Some (List.nth values pos)
  | None -> None

```

This may appear slightly less convenient than association lists (for which Caml offers library routines), but has an advantage which will become clear later: Environment lookup first associates a list index with an identifier, and then references the corresponding value in the values list. Since the list index is only dependent on the lexical structure of the program, a compiler can later compute it statically, therefore reducing environment access in the running code to an addressing mode.

The only required changes to incorporate the new representation of environment into the interpreter are the places where the interpreter looks up the value of a variable in the environment: Here the former application of the environment to the name of the variable needs to be changed to a call to `lookup_env`.

4.8 Introducing Continuation Chains

The continuations in the interpreter for CPS own the important property of being “single threaded”. This means:

- at any point in time, there is one *current continuation*,
- the continuation of the current continuation is the previous current continuation,
- and every continuation is invoked at most once.

This observation suggests replacing the current way of storing continuations in the local environment. Two different ways for storing the continuation that make both use of the single-threadness property come to mind:

- A stack on which the interpreter pushes continuation objects and receives the current continuation by popping it from the stack.
- A linked list of continuation objects where the interpreter keeps a reference to the head of the list.

The first approach corresponds to the classic approach of implementing procedure calls by pushing an activation record on the stack. The second alternative is more flexible as it also extends to languages with support for first-class continuations. Introducing a stack also complicates the memory management. We will therefore stick to the stack-less approach.

An interpreter that supports a chain of linked continuations needs an additional field in the continuation data structure that points to the previous continuation. Analogously, exception handlers need a new field to remember the continuation that was current when `try` installed the handler:

```

type value =
  Unit

```

```

| EmptyList
| Int of int
| Bool of bool
| String of string
| Char of char
| Fun of (value -> handler -> continuation -> value)
| Closure of closure
| Cons of value * value
| Wrong
and closure = environment * ident * ident * exp
and continuation =
  Continuation of environment * handler * ident * continuation * exp
  | Stop
and handler =
  Handler of environment * handler * ident * continuation * exp
  | Error

```

The evaluation function receives an additional parameter for the current continuation and uses this parameter instead of storing the current continuation in the environment:

```

and eval e fenv env h k =
  match e with
  | Return (ki, ve) ->
    let v = eval_val ve fenv env in
    return k v fenv
  | Let (x, ve, e) ->
    eval e fenv (extend_env env x (eval_val ve fenv env)) h k
  | If (ve, e1, e2) ->
    (match eval_val ve fenv env with
     Bool t ->
       if t
       then eval e1 fenv env h k
       else eval e2 fenv env h k)
  | Call(ve1, ve2, c) ->
    let v1 = eval_val ve1 fenv env in
    let v2 = eval_val ve2 fenv env in
    let k = eval_cont c fenv env h k in
    (match v1 with
     Fun f -> f v2 h k
    | Closure (closure_env, x, ki, e) ->
      eval e fenv
        (extend_env closure_env x v2)
        h
        k
    | _ -> return k Wrong fenv)
  | TailCall(ve1, ve2, ki) ->
    let v1 = eval_val ve1 fenv env in
    let v2 = eval_val ve2 fenv env in
    (match v1 with
     Fun f -> f v2 h k
    | Closure (closure_env, x, ki, e) ->
      eval e fenv
        (extend_env closure_env x v2)
        h

```

```

        k
    | _ -> return k Wrong fenv)
| LetCont(ki, c, e) ->
    eval e fenv
    env
    h
    (eval_cont c fenv env h k)

```

The evaluation functions for values and continuations do not change at all:

```

and eval_val ve fenv env =
  match ve with
  | Const c -> eval_const c
  | Ident i ->
    (try env i
     with Not_found -> fenv i)
  | Builtin b -> eval_builtin b fenv
  | Lambda(x, ki, e) ->
    Closure (env, x, ki, e)
and eval_cont (x, e) fenv env h k =
  (Continuation (env, h, x, k, e))

```

Returning to a continuation now fetches the previous continuation from the appropriate field in the continuation data structure:

```

and return k v fenv =
  match k with
  | Stop -> v
  | Continuation (env, h, x, k, e) ->
    eval e fenv (extend_env env x v) h k

```

Evaluation of top-level functions passes the `Stop` continuation as an argument to `eval` instead of binding it in the local environment. The evaluation of programs does not change:

```

let eval_top t fenv =
  match t with
  | WithCont (ki, e) ->
    eval e fenv empty_env
    Error
    Stop
let eval_program (equations, body) =
  let rec fenv f =
    eval_top (List.assoc f equations) fenv
  in
  eval_top body fenv

```

The evaluation function for built-ins remains unchanged except for the exception handling. The code for `try` needs to store the current continuation in the handler but drops storing the continuation in the environment, `raise` extracts the continuation from the `Handler` data structure and passes it to `eval`:

```

let rec eval_builtin i fenv =
  match i with
  | "()" -> Unit
  | "[]" -> EmptyList

```

```

| "true" -> Bool true
| "false" -> Bool false
| "inc" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (match x with
     Int x' -> Int (x' + 1)
     | _ -> Wrong)
   fenv)
| "dec" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (match x with
     Int x' -> Int (x' - 1)
     | _ -> Wrong)
   fenv)
| "zerop" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (match x with
     Int x' -> Bool (x' = 0)
     | _ -> Wrong)
   fenv)
| "+" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (Fun
     (function y -> function h -> function k ->
      return k
        (match x with
         Int x' ->
           (match y with
            Int y' -> Int (x' + y')
            | _ -> Wrong)
         | _ -> Wrong)
        fenv))
     fenv)
| "=" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (Fun
     (function y -> function h -> function k ->
      return k (Bool (x = y)) fenv))
     fenv)
| "::" ->
  Fun
  (function x -> function h -> function k ->
   return k
    (Fun

```

```

        (function y -> function h -> function k ->
          return k (Cons (x, y)) fenv))
      fenv)
| "hd" ->
  Fun
  (function x -> function h -> function k ->
    return k
    (match x with
      Cons (h, _) -> h
    | _ -> Wrong)
    fenv)
| "tl" ->
  Fun
  (function x -> function h -> function k ->
    return k
    (match x with
      Cons (_, t) -> t
    | _ -> Wrong)
    fenv)

| "try" ->
  Fun
  (function thunk -> function h -> function k ->
    return k
    (Fun
      (function handler -> function h -> function k ->
        match thunk with
          Closure (env, x, ki, e) ->
            (match handler with
              Closure (handler_env, handler_x, handler_ki, handler_e) ->
                eval e fenv
                (extend_env env x Unit)
                (Handler (handler_env,
                          h,
                          handler_x, k, handler_e))
                k
              | _ -> return k Wrong fenv)
            | _ -> return k Wrong fenv))
      fenv)
  )
| "raise" ->
  Fun
  (function exc -> function h -> function k ->
    match h with
      Handler (env, h, x, k, e) ->
        eval e fenv (extend_env env x exc) h k
    | Error -> raise Not_found)
  | _ -> raise Not_found

```

Bibliography

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.
- [Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.
- [DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.
- [DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.
- [DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
- [DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.
- [DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.
- [FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.
- [FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
- [Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.
- [Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN’86 Symposium on Compiler Construction.
- [Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.
- [Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

- [Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.
- [PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.
- [PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.
- [Plo75] Gordon Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.
- [SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.
- [ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.
- [ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.
- [WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.