

4.7 Making Functions Machine-Friendly

Now that the representation for environments does not use *function* anymore, 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 $v \rightarrow v$ 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)

```


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.