

TECHNICAL REPORT NO. 303

Semantics-Directed Compilation
of Non-Linear Patterns

by

Olivier Danvy

February 1990

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY

Bloomington, Indiana 47405-4101

Semantics-Directed Compilation of Non-Linear Patterns ¹

Olivier Danvy ²

January 1990

¹Extended version.

²This work has been carried out while the author was visiting the Computer Science Department of Stanford University (thanks to Carolyn L. Talcott) and the Computer Science Department of Indiana University (thanks to Daniel P. Friedman), during the summer and fall of 1989.

This work was supported in part by NSF Grant No. CCR-89-01919.

Abstract

This paper describes the automatic derivation of compiled patterns and of a pattern compiler by partial evaluation. Compiling a pattern is achieved by specializing a pattern matching program with respect to the pattern. Generating a pattern compiler is achieved by specializing the specializer with respect to the pattern matching program, *i.e.*, by self-applying the partial evaluator. The compiled patterns and the compiler are semantics-based because they are obtained using meaning-preserving transformations upon the definitional pattern matching program and the partial evaluator.

The results are unexpectedly good: not only all are the operations depending on the pattern (syntax analysis, resolution of cross-references due to the non-linearity) performed at compile time, but whereas the general pattern matcher builds the substitution environment incrementally and for nothing in case of failure, compiled patterns perform all the structural and equality tests first, and build the result only if the match succeeds. This non trivial runtime staging has been obtained automatically, which is remarkable because staging in general is known to be an art.

This example stresses continuation-passing style as a convenient style for writing general programs. This style makes it possible to circumvent the approximations of binding time analysis (which is an essential requirement for efficient self-application), and in addition, to stage residual programs automatically. These observations have been confirmed by later experiments.

Keywords: partial evaluation, program derivation, compiler generation.

1 Introduction

Pattern matching offers a classical example of a two-argument general program that is worth partially evaluating. Such a program is too general and costly, whereas compiled patterns are tailored to recognizing particular data. This section formalizes pattern matching in lists and compiling patterns. Compiling patterns is achieved naturally using partial evaluation. Similarly a pattern compiler is derived by self-applying the partial evaluator. However self-application requires to analyze the binding times of a source program independently of its actual specialization. Unfortunately, binding time analysis may be too approximative. Some of these approximations turn out to be bypassed using continuation-passing style.

1.1 Pattern matching in lists

We consider pattern matching in proper Lisp lists. A pattern represents a set of lists. Given a pattern and a list, a pattern matching program determines whether the pattern matches the list, *i.e.*, if the list belongs to the set represented by the pattern. The result is either some distinguished mismatch value if the match fails or a collection of substitutions. This traditional problem is addressed at length in books on artificial intelligence programming [Slagle & Gini 87]. It is solved by implementing a program `match` computing the function

$$\mathcal{M} : \text{Pattern} \times \text{List} \rightarrow \text{Unit} + \text{Substitutions}$$

1.2 Compiling patterns

Matching a pattern against a list induces an interpretive overhead – traversing the pattern for each match. This overhead is eliminated by compiling patterns. The associated correctness criterion can be stated as

$$\forall \text{pat} \in \text{Pattern}, \forall \text{dat} \in \text{Data-List}, \mathcal{L} \text{ match}(\text{pat}, \text{dat}) = \mathcal{L} (\mathcal{C}(\text{pat}))(\text{dat})$$

where $\mathcal{L} : \text{Program-Text} \rightarrow \text{Values}^* \rightarrow \text{Answer}$ denotes a language processor and $\mathcal{C} : \text{Pattern} \rightarrow \text{Program-Text}$ denotes the pattern compiler. This compiler can be derived by partial evaluation.

1.3 Partial evaluation

A partial evaluator specializes programs with respect to part of their input. Therefore, compiling a pattern can be achieved by partially evaluating the matching program with respect to the pattern:

$$\forall \text{pat} \in \text{Pattern}, \mathcal{C}(\text{pat}) = S_1^1(\text{match}, \text{pat}) = \mathcal{L} \text{ mix}(\text{match}, \text{pat})$$

where S_1^1 is Kleene's S_n^m function for $n = m = 1$ and `mix` is a program computing S_1^1 , *i.e.*, a partial evaluator. Similarly, deriving a pattern compiler is achieved by specializing the partial evaluator with respect to the matching program:

$$\text{compiler} = S_1^1(\text{mix}, \text{match})$$

where `compiler` is a program computing \mathcal{C} . Experience shows that generating a realistic compiler requires to analyze the binding times of source programs.

1.4 Binding time directed program specialization

Efficient self-application requires to determine the binding times of source expressions in the source program independently of the actual values of its input. For this, the well-known concept of binding times is generalized from variables to expressions. A variable may be bound, say, at compile time, at link time, or at run time. Correspondingly, a source expression may be either reduced at partial evaluation time or at run time. The latter ones need to be rebuilt at partial evaluation time.

As identified in the MIX project, binding times can be safely approximated independently of the actual values of the partial data [Jones *et al.* 89]. As a consequence, source expressions can be classified once and for all by preprocessing, thus avoiding an important interpretive overhead during specialization by cutting down generality. Most of the time, we know which arguments our programs will be specialized with respect to. Presently, we specialize our matching program with respect to a pattern (and not with respect to a data list). Correspondingly we want a compiler for patterns, not for data lists.¹

1.5 An approximation and how to bypass it

Analyzing binding times during preprocessing has a price: often it approximates too coarsely. For example, the result of a conditional expression whose test clause depends on the data list, *i.e.*, that cannot be reduced at partial evaluation time, is excessively approximated, regardless of its then and else clauses.

In this paper, we propose to express source programs tail-recursively to circumvent this approximation by delaying it until the final result. However, because matching requires traversing both pattern and list recursively, a mere iterative style is not enough. We achieve tail-recursion using continuation-passing style.

1.6 Plan and setting

Section 2 describes the partial evaluation of pattern matching in lists. Section 3 analyzes the compilation of patterns. After a comparison with related work, our approach is put into perspective.

The partial evaluators we are using are Anders Bondorf and Olivier Danvy's Similix system [Bondorf & Danvy 90], and Charles Consel's Schism system [Consel 89], both of which analyze the binding times of source programs during preprocessing and are self-applicable. Similix is more automatic but Schism handles partially static structures. We will point out where this makes a difference.

2 Partial Evaluation of Pattern Matching in Lists

2.1 Principle

The general pattern matcher takes a pattern and a datum and traverses them in parallel. By partially evaluating the pattern matcher with respect to the pattern, all accesses, tests, and constructions determined by the pattern are reduced. Pattern constants are inlined in the specialized program, and direct

¹A compiler for data lists maps a list into a program expecting a pattern and determining whether this pattern matches the list. Compiled lists are structurally close to the general matching program – they are obtained by unfolding all control structures depending on the list and reconstructing all the others. Running a compiled list amounts to traversing the pattern recursively in the limits determined by the source list and building a substitution environment as specified by the pattern.

structural and equality tests on the datum are scheduled according to the traversal order of the general pattern matcher.

2.2 Non-linear pattern matching in lists (interpreted mode)

We consider non-linear pattern matching in proper Lisp lists. It is non-linear because pattern variables may occur repeatedly. Lists are formed according to the following grammar:

$$\begin{aligned}\text{List} &= \text{Atom} \mid \text{Pair} \\ \text{Pair} &= \text{List Tail} \\ \text{Tail} &= \text{nil} \mid \text{Pair}\end{aligned}$$

A pattern may be a constant, a variable, or a sequence of patterns. Patterns are formed according to the following BNF:

$$\text{Pat} ::= (\text{Cst Val}) \mid (\text{Var Nam}) \mid (\text{Seq Pat}^*)$$

A constant pattern specifies a constant and matches a list if and only if the list equals this constant. A variable pattern declares a name. The first occurrence of a name is bound to the corresponding list, and later occurrences only match the same list. A sequence of patterns matches a list if the sequence and the list have the same length and each pattern matches the corresponding sublist, recursively. The final result is either a substitution environment or the boolean value `#f`.

For example, the pattern `(Seq (Var x) (Cst 25))` represents the set of all two-element lists whose second element equals 25. A successful match against a list instantiates `x` to the first element of this list. For another example, the pattern `(Seq (Var x) (Var x) (Var x))` represents the set of all three-element lists whose elements are all equal.

Figure 1 displays the source program. Contrarily to most published pattern matching programs in Lisp, it is side-effect free. Contrarily to pattern matching in ML-like functional languages, it is non-linear. It is written in Scheme [Rees & Clinger 86] and uses Scheme's non-logical and control structure (similar to `andalso` in ML) to compute the answer. The program is tail-recursive and traverses the pattern and the datum depth-first, building the substitution environment incrementally.

2.3 Non-linear pattern matching in lists (compiled mode)

Two compiled patterns are displayed in figures 2 and 3. They are iterative because the source program is tail-recursive. All operations on the datum have been scheduled according to the depth-first traversal of the pattern. The compiled patterns are built as a series of conditional expressions embedded with `let` expressions declaring local variables denoting parts of the data list. The `let` expressions have been introduced by the partial evaluator automatically to avoid multiple traversals of the data list. At run time, the data list is traversed depth-first according to the pattern. All structural and equality tests are performed along the traversal and if they all succeed, the substitution environment is built and returned. This environment pairs two lists: a list of names, and a list of values.

These residual programs illustrate a surprising property: whereas the substitution environment was built incrementally in the source program (and thus potentially for nothing in case of failure), *it is built once in compiled programs in case of success and not built at all in case of failure.*

```

Cont = List(Nam) x List(Val) -> Ans      Ans = Pair(List(Nam), List(Val)) + {#f}

(define match   ;;; Pat x Dat -> Ans
  (lambda (p d)
    (dispatch p d empty-lnam empty-lval pair)))

(define dispatch ;;; Pat x Dat x List(Nam) x List(Val) x Cont -> Ans
  (lambda (p d ln lv k)
    (record-case p
      [Cst (v) (and (equal? v d) (k ln lv))]
      [Var (n) (assoc-c n ln lv (lambda (v) (and (equal? d v) (k ln lv)))
                        (lambda () (k (cons-nam n ln) (cons-val d lv))))]
      [Seq p* (match-seq p* d ln lv k)])))

(define match-seq ;;; Pat* x Dat x List(Nam) x List(Val) x Cont -> Ans
  (lambda (p* d ln lv k)
    (destruct-case p*
      [()] (and (null? d) (k ln lv))]
      [(p . p*) (and (pair? d)
                     (dispatch p (car d) ln lv (lambda (ln lv)
                                                  (match-seq p* (cdr d) ln lv k))))]))))

```

Figure 1: Generic pattern matching program.

This program is tail-recursive, continuation-passing. It traverses the pattern depth-first and returns a pair of lists or the boolean value #f. The initial continuation pair is applied if the match succeeds and pairs its two arguments. This pair defines the substitution environment built through matching. The first element of the pair is a list of names, and the second is a list of corresponding values. Each new variable match extends the two lists. Any mismatch interrupts the matching process. The operator `assoc-c` checks the occurrence of a variable earlier in the pattern. It is passed two continuations treating each case, and has the type $Nam \times List(Nam) \times List(Val) \times [Val \rightarrow Ans] \times [Unit \rightarrow Ans] \rightarrow Ans$

```

(define match-0
  (lambda (d_0)
    (and (pair? d_0)
         (let ([d_2_1_ (cdr d_0)])
           (and (pair? d_2_1_)
                (equal? 25 (car d_2_1_))
                (null? (cdr d_2_1_))
                (pair '(x) (list (car d_0))))))))

```

Figure 2: Instance of the generic matching program dedicated to the pattern `(Seq (Var x) (Cst 25))`. This dedicated program traverses the data depth-first, tail-recursively. It is in direct style because all the continuations of the source program depended on the pattern only and thus have been eliminated. Because the list of names is static and the list of values is tail-static, all cross-references have been solved. The result is either a pair of lists or the boolean #f. *However the list of names has been built at compile-time and the list of values is built only if the match succeeds.* This was not the case in the source program and thus illustrates automatic staging of the matching process.

```

(define match-0
  (lambda (d_0)
    (and (pair? d_0)
      (let* ([d_2_0_ (car d_0)] [d_2_1_ (cdr d_0)])
        (and (pair? d_2_1_)
          (let ([d_3_2_ (car d_2_1_)]
                (and (pair? d_3_2_)
                  (let ([d_4_3_ (car d_3_2_)]
                        (and (pair? d_4_3_)
                          (let ([d_5_5_ (cdr d_4_3_)]
                                (and (pair? d_5_5_)
                                  (equal? '(1 2 3) (car d_5_5_))
                                  (let ([d_5_7_ (cdr d_5_5_)]
                                        (and (pair? d_5_7_)
                                          (equal? (car d_5_7_) d_2_0_) ; <---
                                          (null? (cdr d_5_7_))
                                          (let ([d_4_9_ (cdr d_3_2_)]
                                                (and (pair? d_4_9_)
                                                  (null? (cdr d_4_9_))
                                                  (null? (cdr d_2_1_))
                                                  (pair '(z y x)
                                                       (list (car d_4_9_)
                                                            (car d_4_3_)
                                                            d_2_0_)))))))))))))))))))))

```

Figure 3: Result of compiling (Seq (Var x) (Seq (Seq (Var y) (Cst (1 2 3)) (Var x)) (Var z))). The variable x occurs twice, but the corresponding test (cf. arrow) is independent of building the substitution environment. There are no computation duplications because of the let expressions.

2.4 Compilation of patterns

Because control is entirely determined by the pattern, all control constructions, and most notably continuations, are reduced statically. Because the list of names is static, it is built at compile time (and ordered consistently with the depth-first traversal) and inlined.² Because the list of values is *tail-static* (i.e., it is a static list of dynamic values), all cross-references are solved statically.³ What remains is a series of operations over the data list. Because the source program minimized the accesses to the data list, and because our partial evaluators do not duplicate dynamic computations, the residual program accesses the data list minimally and without redundancies. Its correctness stems from the correctness of the source program and of the partial evaluator.

3 Analysis

This section investigates why compiling patterns by partial evaluation is (1) efficient and (2) optimizing.

²For example, the list of names has been built at compile time by accumulating the first occurrences of names in the pattern (cf. figure 3 – the list of names is (z y x), and the first occurrences of names in the pattern were x, y, and z).

³Using Schism, the list of values is an actual list, and using Similix, it is represented procedurally, i.e., using Church pairing.


```

(define match
  (lambda (p d)
    (dispatch p d empty-lnam empty-lval pair)))

(define dispatch
  (lambda (p d ln lv k)
    (record-case p
      [Cst (v) (and (equal? v d) (k ln lv))]
      [Var (n) (assoc-c n ln lv (lambda (v) (and (equal? d v) (k ln lv)))
            (lambda () (k (cons-nam n ln) (cons-val d lv)))]
      [Seq p* (match-seq p* d ln lv k)])))

(define match-seq
  (lambda (p* d ln lv k)
    (destruct-case p*
      [() (and (null? d) (k ln lv))]
      [(p . p*) (and (pair? d)
                      (dispatch p (car d) ln lv (lambda (ln lv)
                                                         (match-seq p* (cdr d) ln lv k)))])))))

```

Figure 4: Annotated pattern matching program.

Underlined expressions depend on the data list. Overlined variables denote partially static values.

$$\text{cons-nam} : \text{Nam} \times \text{List}(\text{Nam}) \rightarrow \text{List}(\text{Nam}) \quad \text{cons-val} : \text{Val} \times \text{List}(\text{Val}) \rightarrow \text{List}(\text{Val})$$

$$\text{assoc-c} : \text{Nam} \times \text{List}(\text{Nam}) \times \text{List}(\text{Val}) \times [\text{Val} \rightarrow \text{Ans}] \times [\text{Unit} \rightarrow \text{Ans}] \rightarrow \text{Ans}$$

3.1 The source program was properly staged

A program is “well-structured” [Emanuelson 80] or “properly staged” [Jørring & Sherlis 86] if the computations depending only on the available data occur first. Computations are represented by expressions in the source program. Binding time analysis classifies these expressions to be *static* (depending only on available data, *i.e.*, reduced at partial evaluation time), or *dynamic* (depending on unavailable data, *i.e.*, reduced at run time and, to this end, rebuilt by the partial evaluator).

Figure 4 displays the source matching program, where all the dynamic expression have been underlined and the variables denoting partially static values have been overlined.

All the bare expressions are static and thus are reduced at partial evaluation time. The pattern is traversed depth-first, and since it is static all the control is statically determined. Therefore all the creations and applications of the continuations are performed at partial evaluation time. The list of values is tail-static: all values are dynamic but the list is built under static control, alongside the list of names which is completely static. Therefore `assoc-c` is unfolded and cross-references are solved.

Compiling a pattern is achieved by unfolding all static and partially static control and data structures and by rebuilding all dynamic expressions, substituting residual expressions for dynamic identifiers. Eventually, the initial continuation is applied to a static list of names and a tail-static list of values, and converts the tail-static list into a residual expression.

3.2 Compiled patterns are properly staged

Originally, the substitution environment is built step by step. It is accessed in three contexts:

- when a new variable is encountered, the environment is extended;
- when a new occurrence of variable is encountered, the offset of this variable in the substitution environment is established, and an equality test is performed;
- when the match succeeds, the environment is returned as a result.

The environment is built by the partial evaluator, symbolically; it is consulted during partial evaluation; and it is returned as a result if the match succeeds. In other terms, the backbone of this data structure lives only at partial evaluation time, even though its slots are filled with dynamic values, *i.e.*, with residual expressions.

The residual program is made out of dynamic expressions that are rebuilt. Relevant references to the data list are spread in the residual program (with `let` expressions) and the actual construction of the result is delayed until the initial continuation is applied: the substitution environment is built only if the match succeeds.

3.3 Direct vs. continuation-passing style

Binding time information is more precise when extracted from a continuation-passing program because the approximation on returned values is delayed until the final result. Therefore the specializer is given more static information on the source program and can perform more computations, yielding better residual programs. Completely static expressions are reduced, yielding static values. Incompletely static expressions are evaluated symbolically, yielding partially static values. Completely dynamic expressions are rebuilt, yielding residual expressions. Partially static values such as the substitution environment can float from their site of creation to where they are actually used, rather than being frozen in the residual program.

4 Single-Threading and η -Reduction

In the program of figure 1, the variables `ln` and `lv` are single-threaded. They are just passed around, only to be accessed in the `assoc-c` procedure or when passed to the initial continuation. Figure 5 displays a curried version of figure 1 where `ln` and `lv` are η -reduced everywhere. The auxiliary procedures are displayed in appendix.

Specializing this program yields the same results as above because it is still in continuation-passing style, and thus still properly staged.

5 Comparison with Related Works

The results reported here compare favorably to existing pattern compilers [Teitelman 78, Peyton Jones 87], which essentially perform partial evaluation by hand. However, the former handles side-effects and the latter addresses linear pattern matching in lazy lists (pointing out how non-linear patterns perturbate laziness) and repeated matches of a datum against several patterns.

```

(define match   ;; Pat x Dat -> Ans
  (lambda (p d)
    ((cur-dispatch p d pair) empty-lnam empty-lval)))

(define cur-dispatch   ;; Pat x Dat x Cont -> Cont
  (lambda (p d k)
    (record-case p
      [Cst (v) (if (equal? v d) k fail)]
      [Var (n) (cur-assoc-c n (test-and-jump d k) (extend n d k))]
      [Seq p* (cur-match-seq p* d k)])))

(define cur-match-seq   ;; Pat* x Dat x Cont -> Cont
  (lambda (p* d k)
    (destruct-case p*
      [()] (if (null? d) k fail)]
      [(p . p*) (if (pair? d)
                    (cur-dispatch p (car d) (cur-match-seq p* (cdr d) k))
                    fail)])))

```

Figure 5: Curried pattern matching program, where `Cont = [List(Nam) x List(Val) -> Ans]`.

Compiling patterns using partial evaluation is a standard exercise, see the bibliography of [Bjørner, Ershov & Jones 88], but the author is not aware of any compilation including an automatic staging. It is generally agreed that source programs should be staged so that the computations depending only on available data can be performed at partial evaluation time. This paper suggests continuation-passing style as a convenient style for programs to be specialized by a self-applicable partial evaluator, to bypass the approximation of the binding time analysis and to achieve automatic staging. This makes it possible to compile patterns *and* to derive the corresponding pattern compiler.

The present work is part of a broader investigation of the actual possibilities of self-applicable partial evaluators. [Consel & Danvy 90] describes the compilation of Algol-like programs where syntax analysis, scope resolution, and static type checking are performed at compile time, by partially evaluating a continuation-passing interpreter. [Consel & Danvy 89] points out how to obtain the effect of the Knuth-Morris-Pratt linear string matching by partially evaluating a naive and quadratic matching program, and the derivation of directed acyclic word graphs is formalized in [Malmkjær & Danvy 90]. There, the key point is to express backtracking under static control so that it is performed at partial evaluation time. Coupling the two techniques – backtracking under static control and continuation-passing style – opens the door to tackling, *e.g.*, pattern matching in trees.

6 Conclusions and Issues

This paper describes the automatic compilation of patterns by partial evaluation of a matching program. The interpretive overhead of traversing the pattern is completely removed. Cross-references due to the non-linearity of patterns are computed at compile-time. Accesses to the data lists are minimized. The substitution environment is built only if the match succeeds, whereas it was built incrementally in the matching program. The compiler is obtained by self-application. As specialized instances, compiled patterns inherit the structure of the pattern matching program. As a specialized instance, the pattern

compiler inherits the structure of the partial evaluator.

This paper also introduces continuation-passing style as a convenient way to structure source programs to make them specialize well. This style circumvents the approximations of binding time analysis by delaying them until the final result and as a byproduct, seems propitiative for dynamic staging. Thus conversion into continuation-passing style appears to be an interesting rewriting method for staging source programs.

Acknowledgements

Thanks are due to Karoline Malmkjær, Charles Consel, and Andrzej Filinski for commenting earlier versions of this paper.

References

- [Bjørner, Ershov & Jones 88] Dines Bjørner, Andrei P. Ershov, Neil D. Jones (eds.): *Partial Evaluation and Mixed Computation*, North-Holland (1988)
- [Bondorf & Danvy 90] Anders Bondorf, Olivier Danvy: *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types*, DIKU report 90-04, University of Copenhagen, Denmark (1989)
- [Consel & Danvy 89] Charles Consel, Olivier Danvy: *Partial Evaluation of Pattern Matching in Strings*, Information Processing Letters, Vol. 30, No 2 pp 79-86 (1989)
- [Consel 89] Charles Consel: *Analyse de Programme, Evaluation Partielle, et Génération de Compilateurs*, PhD thesis, University of Paris VI, France (June 1989)
- [Consel & Danvy 90] Charles Consel, Olivier Danvy: *Static and Dynamic Semantics Processing*, Technical Report 761, Computer Science Department, Yale University (November 1989)
- [Emanuelson 80] Pär Emanuelson: *Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation*, PhD thesis, Linköping University, Sweden (1980)
- [Jones et al. 89] Neil D. Jones, Peter Sestoft, Harald Søndergaard: *MIX: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, Vol. 2, No 1 pp 9-50 of the International Journal LISP and Symbolic Computation (1989)
- [Jørring & Sherlis 86] Ulrik Jørring, William L. Scherlis: *Compilers and Staging Transformations*. proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages pp 86-96, St. Petersburg, Florida (1986)
- [Malmkjær & Danvy 90] Karoline Malmkjær, Olivier Danvy: *Preprocessing by Specialization*, Technical Report TR-CS-90-3, Department of Computer and Information Sciences, Kansas State University (October 1989)
- [Peyton Jones 87] Simon L. Peyton Jones: *The Implementation of Functional Programming Languages*, Prentice-Hall (1987)
- [Rees & Clinger 86] Jonathan Rees, William Clinger (eds.): *Revised³ Report on the Algorithmic Language Scheme*, Sigplan Notices, Vol. 21, No 12 pp 37-79 (December 1986)
- [Slagle & Gini 87] James R. Slagle, Maria L. Gini: *Pattern Matching*, pp 716-720 of *Encyclopedia of Artificial Intelligence*, Stuart C. Shapiro (ed.), Wiley-Interscience (1987)

```
(define fail ;; Cont
  (lambda (ln lv)
    #f))

(define test-and-jump ;; Val x Cont -> Val -> Cont
  (lambda (d k)
    (lambda (v)
      (if (equal? d v)
          k
          fail))))

(define extend ;; Nam x Val x Cont -> Cont
  (lambda (n d k)
    (lambda (ln lv)
      (k (cons-nam n ln) (cons-val d lv)))))

(define cur-assoc-c ;; Nam x [Val -> Cont] x [Unit -> Cont] -> Cont
  (lambda (n s f)
    (lambda (ln lv)
      (let ([offset (index n ln)])
        (if (negative? offset)
            ((f) ln lv)
            ((s (list-ref lv offset)) ln lv))))))

(define index ;; Nam x List(Nam) -> Nat + {-1}
  (lambda (e l)
    ((rec loop (lambda (n l)
                 (cond
                  [(null? l)
                   -1]
                  [(equal? e (car l))
                   n]
                  [else
                   (loop (add1 n) (cdr l))])))) 0 l)))
```

Figure 6: Miscellaneous.