Refunctionalization at Work

Olivier Danvy
Kevin Millikin

BRICS Report Series RS-07-7

ISSN 0909-0878 March 2007

Refunctionalization at Work[*]

Olivier Danvy and Kevin Millikin
BRICS
Department of Computer Science
University of Aarhus[†]

March 21, 2007

Abstract

We present the left inverse of Reynolds's defunctionalization and we show its relevance to programming and to programming languages. We present two methods to put a program that is almost in defunctionalized form into one that is actually in defunctionalized form, and we illustrate them with a recognizer for Dyck words and with Dijkstra's shunting-yard algorithm.

[*]This article documents an invited talk at MPC '06 [20].
[†]IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `<danvy@brics.dk>`, `<kmillikin@brics.dk>`

Contents

List of Figures

1 Introduction

This article is a continuation of Danvy and Nielsen's earlier article "Defunctionalization at Work" [25], that outlined the extent to which Reynolds's defunctionalization [41] is pervasive in writing and transforming programs, and in specifying and implementing programming languages. Our goal here is to show that the left inverse of defunctionalization, i.e., *re*functionalization, is also relevant to programming and to programming languages.

1.1 Defunctionalization: origin, motivation, and applications

Reynolds introduced defunctionalization to specify higher-order definitional interpreters using first-order means [41]. He presented it as a mere programming technique, and, except for deriving a first-order semantics in his textbook on programming languages [44, Section 12.4], he never used it again [43]. Since then, defunctionalization has been chiefly used in compilers [12, 14], program transformers [47], and partial evaluators [11, 15]. It has also been independently discovered in the context of logic programming [49], and subsequently it has been formalized in a typed setting [5, 6, 38, 40].

Over the last few years [1, 8, 19, 36, 37], Danvy and his students have retraced Reynolds's steps from higher-order to first-order interpreter by closure conversion (to make the data flow first order), transformation into Continuation-Passing Style (to sequentialize the control flow), and defunctionalization (to make the control flow first order). They have identified not only that a defunctionalized, CPS-transformed, and closure-converted interpreter has the structure of an abstract machine, but also that a large number of independently designed abstract machines for variants of the λ-calculus are the defunctionalized, CPS-transformed, and closure-converted counterpart of a compositional interpreter [2–4, 7, 9, 18, 24], including Felleisen et al.'s CEK machine, which was actually designed as such [27, page 196]. A practical byproduct of this observation is that evaluation contexts—which are notoriously non-trivial to get right—can be obtained mechanically by defunctionalizing the continuations of a compositional interpreter.

1.2 Refunctionalization

Not all abstract machines, however, are in defunctionalized form. It is our thesis here that a number of them can be restated to be so. The goal of this article is to present two techniques for restating programs that are "almost" in defunctionalized form to make them *be* in defunctionalized form: disentangling and merging apply functions (see Section 3). These programs can then be refunctionalized into a higher-order—and more often than not, continuation-based—counterpart.

Of course, not all defunctionalized programs and not all refunctionalized programs are interesting independently of each other, but some of them are. We mentioned above the case of abstract machines for the λ-calculus. Another example is the samefringe problem: McCarthy's famous simple solution [35] and, e.g., Henderson and Morris's iterative solution based on lazy lists [31] are the defunctionalized / refunctionalized counterparts of each other [10, Section 5]. Another example is the reverse function: Hughes's efficient reverse function based on curried list constructors [32] and the usual accumulator-based fast reverse function are the defunctionalized / refunctionalized counterparts of each other [25]. More examples are available elsewhere [19, 25, 28, 36, 37] as well as in Sections 4 and 5, but let us mention a last one that pertains to refunctionalization. On the basis that continuations are a *functional* representation of control in a continuation-passing evaluation function, Landin is not listed among the co-discoverers of continuations [42]. Yet his SECD machine is in defunctionalized form and the defunctionalized counterpart of a compositional continuation-passing evaluation function [18]. Furthermore, the J operator captures the current dump, i.e., in refunctionalized form, the continuation of the caller [24]. This observation has led us to suggest that Landin's name be added to the list of co-discoverers of continuations [24].

1.3 Overview

We first review defunctionalization and its left inverse, refunctionalization (Section 2). We then present two techniques for restating programs from being almost in defunctionalized form to being in defunctionalized form: disentangling and merging apply functions (Section 3), and we illustrate these techniques with two examples: recognizing Dyck words (Section 4) and Dijkstra's shunting-yard algorithm (Section 5). After reviewing other applications (Section 6), we conclude (Section 7).

Prerequisites: We expect a basic familiarity with Standard ML,[1] with continuation-passing style (CPS) [21, 39, 46], and with the left inverse of the CPS transformation [23]. The presentation of defunctionalization below should be self-contained, but the reader should not deny himself the pleasure of re-reading “Definitional Interpreters” [41].

2 Refunctionalization: the left inverse of defunctionalization

In his study of definitional interpreters for programming languages [41], Reynolds drew a distinction between those that use higher-order functions, and thus possibly depend on the scoping rules of their metalanguage, and those that use only first-order data structures, and thus are independent of the scoping rules of their metalanguage. He introduced defunctionalization to transform programs that use higher-order functions into ones that use first-order data structures. Refunctionalization is the inverse transformation. It transforms programs that use first-order data structures into ones that use higher-order functions.

2.1 Defunctionalization

Intuitively, defunctionalization replaces a function type with a polynomial (sum of products) data type, and introduces an apply function that interprets the data type given the argument values of the original function type. It replaces abstractions creating elements of the function type with applications of the data-type constructors. It replaces applications of values of the function type to arguments with an application of the apply function to a value of the first-order type and the same arguments. The defunctionalization algorithm can be informally sketched as follows.

The algorithm assumes that one has identified a target set of function abstractions. In a typed language, these abstractions minimally all have the same type. Reynolds identified sets of abstractions by their “types” according to the semantic domains of a definitional interpreter. Later work [5, 6, 25, 38, 40] considered all functions of a given type, or sets of functions determined by a control-flow analysis [45]. In any case, the set has to be closed

[1]And with the option data type: `datatype 'a option = NONE | SOME of 'a`.

under data flow to call sites: if a function in the set can be called from a call site in the program, then *all* functions that can be called from that call site are in the set.

We assume the identification of a set of n abstractions $\{\lambda x.M_i \mid 1 \leq i \leq n\}$ all of type $\tau \to \tau'$, and closed under data flow to call sites.[2] Each abstraction has a (possibly empty) set of m_i typed free variables $\{x^i_j : \tau^i_j \mid 1 \leq j \leq m_i\}$, not including variables that are bound in the top-level environment. Defunctionalization consists of performing all of the following steps:

1. **Introduce a first-order data type:** There are n data-type constructors C_i for $1 \leq i \leq n$. The constructors uniquely identify the abstractions being defunctionalized, and each variant of the data type represents the abstractions. There are thus n variants and variant i represents the values of the variables occurring free in abstraction i:

$$\begin{array}{rcl} \textit{datatype } \tau\_\textit{arrow}\_\tau' & = & C_1 \textit{ of } \tau^1_1 \times \ldots \times \tau^1_{m_1} \\ & | & \ldots \\ & | & C_n \textit{ of } \tau^n_1 \times \ldots \times \tau^n_{m_n} \end{array}$$

   Introduce an apply function: The apply function dispatches on the data-type constructor to determine which abstraction is being applied and to bind its free variables:

$$\begin{array}{l} \textit{fun apply}(f, x) = \textit{case } f \textit{ of} \\ \qquad\qquad\quad C_1(x^1_1, \ldots, x^1_{m_1}) \Rightarrow M_1 \\ \qquad\qquad \mid \ \ldots \\ \qquad\qquad \mid \ C_n(x^n_1, \ldots, x^n_{m_n}) \Rightarrow M_n \end{array}$$

2. **Replace abstractions:** An abstraction $\lambda x.M_i$ with free variables $\{x^i_j : \tau^i_j \mid 1 \leq j \leq m_i\}$ is replaced with an application of the data-type constructor, $C_i(x^i_1, \ldots, x^i_{m_i})$.

   Replace applications: Any call site $f\ x$ that may be an application of one of the abstractions is replaced with a call to the apply function, *apply*(f, x).

2.2 Refunctionalization

Refunctionalization is the left inverse of defunctionalization, mapping data types and apply functions in the image of defunctionalization back to higher-order functions. A program with a first-order data structure and an apply function dispatching on it is in the image of Reynolds's defunctionalization algorithm if this apply function is the only one dispatching on the data type. More generally, if there is syntactically only a single case dispatch on the data type, then the dispatch can be abstracted into an apply function whose arguments are the free variables of the entire case expression and whose return type is the type of the case expression, as described in Section 3.1.

Once a data type and an apply function are recognized as being in defunctionalized form, refunctionalization proceeds by reversing the steps of defunctionalization. A data type δ with an apply function of type $\delta \times \tau \to \tau'$ can be refunctionalized into the functional type $\tau \to \tau'$. Refunctionalization consists of performing all of the following steps:

1. **Replace applications of the apply function:** A call to the apply function, *apply*(f, x), is replaced with the call to the applied function, $f\ x$.

[2]For simplicity we consider single-argument functions. Without loss of generality, defunctionalization and refunctionalization can be applied to programs with multiple-argument, curried or uncurried, functions as well.

Replace data-type constructor applications: Data-type constructor applications are replaced with abstractions based on the apply function. We assume an apply function:

$$\begin{aligned} \textit{fun apply}(f, x) = \textit{case}\ f\ \textit{of}\ & C_1(x^1_1, \ldots, x^1_{m_1}) \ \Rightarrow M_1 \\ \mid\ & \ldots \\ \mid\ & C_n(x^n_1, \ldots, x^n_{m_n}) \Rightarrow M_n \end{aligned}$$

Each constructor application $C_i(v_1, \ldots, v_{m_i})$ of C_i applied to values v_j for $1 \leq j \leq m_i$ is replaced by $(\lambda x.M)\{v_1/x^i_1, \ldots, v_{m_i}/x^i_{m_i}\}$, the capture-avoiding substitution of the constructor arguments for the free variables in the abstraction represented by the apply function. If a constructor is applied to non-values, we insert let-bindings for the non-value arguments before performing this step.

2. **Remove the apply function:** Because the apply function is never called, its definition is no longer needed.

   Remove the data-type definition: Because the constructors of the data type are never used, and the only case dispatch on them (the apply function) has been removed, the data-type definition is no longer needed.

Refunctionalization is akin to the Scott-encoding [48] of a data type, i.e., the functional representation of a data type as its case-dispatch function.

3 Towards putting programs in refunctionalized form

The hallmark of a defunctionalized data type and an apply function is that the apply function is the single point of consumption for the data type. Disentangling (Section 3.1) abstracts data-type consumptions into functions, and merging (Section 3.2) combines multiple candidate apply functions for the same data type when possible. Both are illustrated in Sections 4 and 5.

3.1 Disentangling

A program can fail to be in the image of defunctionalization if there are multiple points of consumption for elements of a data type[3] or if the single point of consumption is not in a separate function.

‘Disentangling’ consists of abstracting each case expression dispatching on a data type into a separate function. The arguments of the function are the free variables of the case expression and their types are the types of the free variables. The return type is the type of the case expression.

In the case of abstract machines, disentangling the transition function consists of splitting single transitions that dispatch simultaneously on multiple components of the state into multiple serial transitions that dispatch separately on single components of the state.

3.2 Merging apply functions

After disentangling, every case dispatch on a data type is abstracted into a function which is a candidate apply function. If there is a single apply function, then the program can be

[3]This can occur, for example, when a program is defunctionalized and then the apply function is inlined.

refunctionalized. If there are multiple apply functions, then they can be merged under some conditions.

A technique that frequently works for abstract machines is to merge the apply functions using the universal property of sum types. Specifically, two functions $\mathit{apply}_1 : \delta \times \tau_1 \to \tau'$ and $\mathit{apply}_2 : \delta \times \tau_2 \to \tau'$ can be merged into a single function $\mathit{apply} : \delta \times (\tau_1 + \tau_2) \to \tau'$ that performs a case dispatch on its second argument. To reflect this merging, calls to apply_1 and apply_2 are adjusted to call *apply* with their second argument injected into the appropriate summand.

This technique works when the return types of the possible apply functions are all the same. In abstract machines this is usually the case, because the transition functions are all tail-recursive and so they share the same return type.

4 A worked-out example: recognizing Dyck words

Dyck words are well-balanced strings of left and right parentheses, which we represent in ML as follows:

```
datatype parenthesis = L | R

type word = parenthesis list
```

For example, the list `[L, L, R, L, R, R]` forms a Dyck word whereas the list `[R, L]` does not.

Dyck words are classically recognized with an abstract machine implementing a push-down automaton. This state-transition system operates iteratively over a given list and a counter reflecting the number of open parentheses seen so far in the list:

```
datatype nat = ZERO | SUCC of nat
```

The machine starts with a given word and a zero counter. At each iteration, one of the following transitions takes place:

- if the list of parentheses is empty and the counter is zero, a final, accepting state is reached;
- if the list of parentheses is empty and the counter is positive, a final, non-accepting state is reached;
- if the first parenthesis is a left one, the tail of the list is taken and the counter is incremented;
- if the first parenthesis is a right one and if the counter is zero, a final, non-accepting state is reached;
- if the first parenthesis is a right one and if the counter is positive, the tail of the list is taken and the counter is decremented.

The following ML program implements this transition system:

```
(*  recognize : word -> bool  *)
fun recognize ps
    = let (*  run : word * nat -> bool  *)
          fun run ([], ZERO)
              = true
            | run ([], SUCC c)
              = false
            | run (L :: ps, c)
              = run (ps, SUCC c)
            | run (R :: ps, ZERO)
              = false
            | run (R :: ps, SUCC c)
              = run (ps, c)
      in run (ps, ZERO)
      end
```

The goal of this section is to refunctionalize this program with respect to the counter.

In the program above, both parameters of `run` serve as induction variables: `run` dispatches both over the list of parentheses and over the counter. We therefore disentangle `run` by introducing two specialized, mutually recursive versions: one with respect to an empty word (`run_nil`) and the other with respect to a right parenthesis (`run_par`). The resulting disentangled program is displayed in Figure 1: `run` solely dispatches over the list of parentheses, and `run_nil` and `run_par` solely dispatch over the counter.

The transition system displayed in Figure 1 operates in lockstep with the original transition system,[4] but it is not in defunctionalized form with respect to the counter: the counter is dispatched upon both in `run_nil : nat -> bool` and in `run_par : nat * word -> bool`. We therefore merge `run_nil` and `run_par` into a function `run_aux : nat * word option -> bool`. The resulting merged program is displayed in Figure 2. It is in defunctionalized form.

The refunctionalized counterpart of the merged program in Figure 2 is displayed in Figure 3. Its function `word option -> bool` is the refunctionalized counterpart of the data type `nat` and the apply function `run_aux`. This program is in CPS, and except for errors, it uses its continuations linearly and in order. Its direct-style counterpart is below. It is a recursive program where not all calls are in tail position. As an aid to the eye, we have shaded the non-tail calls in grey. Errors are handled with `callcc` and `throw` [23]:

```
fun recognize_in_direct_style ps
    = callcc (fn exit => let (*  run : word -> word option  *)
                             fun run []
                                 = NONE
                               | run (L :: ps)
                                 = (case run ps
                                      of NONE    => throw exit false
                                       | SOME ps => run ps)
                               | run (R :: ps)
                                 = SOME ps
                         in case run ps
                              of NONE    => true
                               | SOME ps => false
                         end)
```

[4]The new machine takes one or two steps for each step in the original one.

```
fun recognize_disentangled ps
    = let (*  run : word * nat -> bool  *)
          fun run ([], c)
              = run_nil c
            | run (L :: ps, c)
              = run (ps, SUCC c)
            | run (R :: ps, c)
              = run_par (c, ps)
          (*  run_nil : nat -> bool  *)
          and run_nil ZERO
              = true
            | run_nil (SUCC n)
              = false
          (*  run_par : nat * word -> bool  *)
          and run_par (ZERO, ps)
              = false
            | run_par (SUCC c, ps)
              = run (ps, c)
      in run (ps, ZERO)
      end
```

Figure 1: Dyck-words recognizer: disentangled version

```
fun recognize_merged ps
    = let (*  run : word * nat -> bool  *)
          fun run ([], c)
              = run_aux (c, NONE)
            | run (L :: ps, c)
              = run (ps, SUCC c)
            | run (R :: ps, c)
              = run_aux (c, SOME ps)
          (*  run_aux : word * word option -> bool  *)
          and run_aux (ZERO, NONE)
              = true
            | run_aux (ZERO, SOME ps)
              = false
            | run_aux (SUCC c, NONE)
              = false
            | run_aux (SUCC c, SOME ps)
              = run (ps, c)
      in run (ps, ZERO)
      end
```

Figure 2: Dyck-words recognizer: merged version

```
fun recognize_refunctionalized ps
    = let (*  run : word * (word option -> bool) -> bool  *)
          fun run ([], c)
              = c NONE
            | run (L :: ps, c)
              = run (ps, fn NONE => false | SOME ps => run (ps, c))
            | run (R :: ps, c)
              = c (SOME ps)
      in run (ps, fn NONE => true | SOME ps => false)
      end
```

Figure 3: Dyck-words recognizer: refunctionalized version

The counter that was explicit in the original specification above, the disentangled one (Figure 1), the merged one (Figure 2), and the refunctionalized one (Figure 3) is now implicit in this direct-style program. In other words, the original recognizer was implemented by a tail-recursive push-down automaton that managed an explicit data stack—namely the counter. This explicit data stack is now implicit in the control stack of the language processor for this recursive program in direct style.

5 A worked-out example: Dijkstra's shunting-yard algorithm

The shunting-yard algorithm is used to parse an arithmetic expression from infix form (as a stream of tokens: literals, operators with precedence, and parentheses) to postfix form (again, as a stream of tokens) or to an abstract-syntax tree. We consider the latter here:

```
datatype token = LIT of int | ADD | MUL | L_P | R_P

datatype expression = INT of int
                    | PLUS of expression * expression
                    | TIMES of expression * expression
```

For example, the ML program implementing the algorithm maps the list of tokens

```
[LIT 10, MUL, LIT 20, ADD, LIT 30, MUL, L_P, LIT 40, ADD, LIT 50, R_P]
```

into the abstract-syntax tree

```
PLUS (TIMES (INT 10, INT 20), TIMES (INT 30, PLUS (INT 40, INT 50)))
```

that represents $10 \times 20 + 30 \times (40 + 50)$. Using `++` and `**` as ML infix notation for `PLUS` and `TIMES`, the result reads `((INT 10) ** (INT 20)) ++ ((INT 30) ** ((INT 40) ++ (INT 50)))`. We make use of this infix notation in Figures 4, 5, and 6 to construct lists of expressions.

The algorithm is defined with an abstract machine implementing a pushdown automaton with two stacks. This state-transition system operates iteratively over a stack of subtrees and a stack of operators and parenthetical separators:

```
datatype operator = XADD | XMUL | XPAR
```

The machine starts with a given list of tokens, an empty stack of subtrees, and an empty stack of operators. At each iteration, one of the following transitions takes place:

- if the first token of the input list is a literal (`LIT n`), the tail of the list is taken and the corresponding expression (`INT n`) is pushed on the stack of subtrees;
- if the first token of the input list is an operator x (`ADD` or `MUL`), then
  - if the stack of operators is empty, the operator on top of this stack is a parenthesis (`XPAR`), or its precedence is strictly lower than that of x, the tail of the input list is taken and the operator corresponding to x (`XADD` or `XMUL`) is pushed on the stack of operators;
  - otherwise, the precedence of the operator on top of the stack is higher than or the same as that of x; this operator is popped off the stack, two expressions are popped off the stack of subtrees, the corresponding expression combining the popped operator and the popped expressions is pushed back, and the operator corresponding to x is pushed on the stack;

- if the first token is a left parenthesis (L_P), the tail of the input list is taken and the corresponding operator (XPAR) is pushed on the stack;
- if the first token is a right parenthesis (R_P), operators are popped off the stack (and for each of them, two expressions are popped off the stack of subtrees and the corresponding expression combining the popped operator and the popped expressions is pushed back) until a parenthesis is met on the stack of operators; the tail of the input list is taken and the parenthesis is popped from the stack of operators;
- otherwise, a final, non-accepting state is reached.

When the list of tokens is exhausted, the stack of operators is iteratively emptied in concert with the stack of subtrees as described above. The final result is the expression on top of the stack of subtrees.

Figure 4 displays a program written in Standard ML that implements the shunting-yard algorithm. The goal of this section is to refunctionalize this program with respect to its stack of operators.

As an ML program, the transition system in Figure 4 is difficult to read because several of the parameters of `run` serve as induction variables: `run` dispatches both over the list of tokens and over the stack of operators. (In contrast, the stack of subtrees is only threaded passively and, except for errors in the input list of tokens, it does not influence the control flow of the program.)

We therefore disentangle `run` into several specialized, mutually recursive versions: one with respect to an empty stack of operators (`run_nil`), three with respect to each of the possible operators on top of the stack (`run_add`, `run_mul`, and `run_par`), and one to dispatch on the stack of operators (`run_aux`). The resulting disentangled program is displayed in Figure 5: all of `run_nil`, `run_add`, `run_mul`, and `run_par` solely dispatch over the list of tokens, and `run_aux` solely dispatches over the stack of operators.

In addition to operating in lockstep with the original transition system implemented in Figure 4,[5] the transition system implemented in Figure 5 is also in defunctionalized form: the stack of operators and `run_aux` respectively form a data type and an apply function that are in the image of defunctionalization.

As for all transition systems in defunctionalized form, the refunctionalized counterpart of the program in Figure 5 is in CPS. Furthermore—again save for errors in the input list of tokens—it uses its continuations linearly and in order, and can therefore be expressed in direct style. We spare the reader with this CPS program, and display the corresponding direct-style program in Figure 6: it is a recursive program where not all calls are in tail position. As an aid to the eye, we have shaded the non-tail calls in grey. Errors are handled with `callcc` and `throw` [23]

The stack of operators that was explicit in Figures 4 and 5 is now implicit in the direct-style program of Figure 6. In other words, the original algorithm was implemented by a tail-recursive automaton with two stacks. The stack of operators is now implicit in the control stack of the language processor for this recursive program in direct style.

The refunctionalized shunting-yard algorithm clearly exhibits the standard features of a bottom-up parser: a return corresponds to a reduce action, a simple tail call with trivial arguments corresponds to a shift action, and a non-tail call corresponds to a state transition where the control-flow at return time implements the goto transition associated to a reduce action.

[5]The new machine takes one or two steps for each step in the original one.

```
fun parse ts = let
      (*  run : token list * expression list * operator list -> expression option  *)
      fun run (       ts as [],       e :: [],               []) = SOME e
        | run (       ts as [], e2 :: e1 :: es,       XADD :: xs) = run (ts, e1 ++ e2 :: es,         xs)
        | run (       ts as [], e2 :: e1 :: es,       XMUL :: xs) = run (ts, e1 ** e2 :: es,         xs)
        | run (    LIT n :: ts,             es,               xs) = run (ts,    INT n :: es,         xs)
        | run (      ADD :: ts,             es,               []) = run (ts,             es, XADD :: [])
        | run (      ADD :: ts,             es, xs as XPAR ::  _) = run (ts,             es, XADD :: xs)
        | run (ts as ADD ::  _, e2 :: e1 :: es,       XADD :: xs) = run (ts, e1 ++ e2 :: es,         xs)
        | run (ts as ADD ::  _, e2 :: e1 :: es,       XMUL :: xs) = run (ts, e1 ** e2 :: es,         xs)
        | run (      MUL :: ts,             es,               []) = run (ts,             es, XMUL :: [])
        | run (      MUL :: ts,             es, xs as XPAR ::  _) = run (ts,             es, XMUL :: xs)
        | run (      MUL :: ts,             es, xs as XADD ::  _) = run (ts,             es, XMUL :: xs)
        | run (ts as MUL ::  _, e2 :: e1 :: es,       XMUL :: xs) = run (ts, e1 ** e2 :: es,         xs)
        | run (      L_P :: ts,             es,               xs) = run (ts,             es, XPAR :: xs)
        | run (      R_P :: ts,             es,       XPAR :: xs) = run (ts,             es,         xs)
        | run (ts as R_P ::  _, e2 :: e1 :: es,       XADD :: xs) = run (ts, e1 ++ e2 :: es,         xs)
        | run (ts as R_P ::  _, e2 :: e1 :: es,       XMUL :: xs) = run (ts, e1 ** e2 :: es,         xs)
        | run (             ts,             es,               xs) = NONE
    in run (ts, [], [])
    end
```

Figure 4: Dijkstra's shunting-yard algorithm in ML

```
fun parse ts = let
      fun run_nil (         [], e :: []) = SOME e
        | run_nil (LIT n :: ts,      es) = run_nil (ts, INT n :: es)
        | run_nil (  ADD :: ts,      es) = run_add (ts,          es, [])
        | run_nil (  MUL :: ts,      es) = run_mul (ts,          es, [])
        | run_nil (  L_P :: ts,      es) = run_par (ts,          es, [])
        | run_nil (         ts,      es) = NONE

      and run_add (       ts as [], e2 :: e1 :: es, xs) = run_aux (ts, e1 ++ e2 :: es,         xs)
        | run_add (    LIT n :: ts,             es, xs) = run_add (ts,    INT n :: es,         xs)
        | run_add (ts as ADD ::  _, e2 :: e1 :: es, xs) = run_aux (ts, e1 ++ e2 :: es,         xs)
        | run_add (      MUL :: ts,             es, xs) = run_mul (ts,             es, XADD :: xs)
        | run_add (      L_P :: ts,             es, xs) = run_par (ts,             es, XADD :: xs)
        | run_add (ts as R_P ::  _, e2 :: e1 :: es, xs) = run_aux (ts, e1 ++ e2 :: es,         xs)
        | run_add (             ts,             es, xs) = NONE

      and run_mul (       ts as [], e2 :: e1 :: es, xs) = run_aux (ts, e1 ** e2 :: es,         xs)
        | run_mul (    LIT n :: ts,             es, xs) = run_mul (ts,    INT n :: es,         xs)
        | run_mul (ts as ADD ::  _, e2 :: e1 :: es, xs) = run_aux (ts, e1 ** e2 :: es,         xs)
        | run_mul (ts as MUL ::  _, e2 :: e1 :: es, xs) = run_aux (ts, e1 ** e2 :: es,         xs)
        | run_mul (      L_P :: ts,             es, xs) = run_par (ts,             es, XMUL :: xs)
        | run_mul (ts as R_P ::  _, e2 :: e1 :: es, xs) = run_aux (ts, e1 ** e2 :: es,         xs)
        | run_mul (             ts,             es, xs) = NONE

      and run_par (    LIT n :: ts,             es, xs) = run_par (ts,    INT n :: es,         xs)
        | run_par (      ADD :: ts,             es, xs) = run_add (ts,             es, XPAR :: xs)
        | run_par (      MUL :: ts,             es, xs) = run_mul (ts,             es, XPAR :: xs)
        | run_par (      L_P :: ts,             es, xs) = run_par (ts,             es, XPAR :: xs)
        | run_par (      R_P :: ts,             es, xs) = run_aux (ts,             es,         xs)
        | run_par (             ts,             es, xs) = NONE

      and run_aux (ts, es,         []) = run_nil (ts, es)
        | run_aux (ts, es, XADD :: xs) = run_add (ts, es, xs)
        | run_aux (ts, es, XMUL :: xs) = run_mul (ts, es, xs)
        | run_aux (ts, es, XPAR :: xs) = run_par (ts, es, xs)
    in run (ts, [], [])
    end
```

Figure 5: The shunting-yard algorithm in defunctionalized form

```
fun parse ts =
    callcc (fn exit => let fun run_nil (             [],        e :: []) = SOME e
                             | run_nil (    LIT n :: ts,             es) = run_nil (ts, INT n :: es)
                             | run_nil (      ADD :: ts,             es) = run_nil (run_add (ts, es))
                             | run_nil (      MUL :: ts,             es) = run_nil (run_mul (ts, es))
                             | run_nil (      L_P :: ts,             es) = run_nil (run_par (ts, es))
                             | run_nil (             ts,             es) = NONE

                           and run_add (       ts as [], e2 :: e1 :: es) = (ts, e1 ++ e2 :: es)
                             | run_add (    LIT n :: ts,             es) = run_add (ts, INT n :: es)
                             | run_add (ts as ADD ::  _, e2 :: e1 :: es) = (ts, e1 ++ e2 :: es)
                             | run_add (      MUL :: ts,             es) = run_add (run_mul (ts, es))
                             | run_add (      L_P :: ts,             es) = run_add (run_par (ts, es))
                             | run_add (ts as R_P ::  _, e2 :: e1 :: es) = (ts, e1 ++ e2 :: es)
                             | run_add (             ts,             es) = throw exit NONE

                           and run_mul (       ts as [], e2 :: e1 :: es) = (ts, e1 ** e2 :: es)
                             | run_mul (    LIT n :: ts,             es) = run_mul (ts, INT n :: es)
                             | run_mul (ts as ADD ::  _, e2 :: e1 :: es) = (ts, e1 ** e2 :: es)
                             | run_mul (ts as MUL ::  _, e2 :: e1 :: es) = (ts, e1 ** e2 :: es)
                             | run_mul (      L_P :: ts,             es) = run_mul (run_par (ts, es))
                             | run_mul (ts as R_P ::  _, e2 :: e1 :: es) = (ts, e1 ** e2 :: es)
                             | run_mul (             ts,             es) = throw exit NONE

                           and run_par (    LIT n :: ts,             es) = run_par (ts, INT n :: es)
                             | run_par (      ADD :: ts,             es) = run_par (run_add (ts, es))
                             | run_par (      MUL :: ts,             es) = run_par (run_mul (ts, es))
                             | run_par (      L_P :: ts,             es) = run_par (run_par (ts, es))
                             | run_par (      R_P :: ts,             es) = (ts, es)
                             | run_par (             ts,             es) = throw exit NONE
                       in run_nil (ts, [])
                       end)
```

Figure 6: The shunting-yard algorithm, refunctionalized and expressed in direct style

6 Other applications

In three situations we had to intervene to put an abstract machine into defunctionalized form:

The SECD machine: The SECD machine only needed to be disentangled to be put in defunctionalized form [18].

The SECD machine with the J operator: Two versions of the SECD machine with the J operator exist—the original one by Landin and Burge [13] and a version due to Felleisen [26]. Disentangling Felleisen's version yields a machine that is in defunctionalized form, but this is not so for Landin and Burge's version. We have put it in defunctionalized form by merging its apply functions [24].

Strong-reduction strategies: In our ongoing work on strongly reducing abstract machines [37], we use both disentangling and merging to refunctionalize a number of abstract machines [16, 17, 29, 33, 34] into compositional normalization functions.

7 Conclusion

We have outlined the extent to which the left inverse of Reynolds's defunctionalization is relevant to transforming programs and to implementing programming languages. Elsewhere, we have illustrated its relevance to programming [10, 22, 25] and to specifying programming languages [2–4, 7, 9, 18, 24].

In some sense, our work on defunctionalization and refunctionalization provides a concrete illustration for the paragraph that follows the definition of the quicksort algorithm, in Section 2.3.3 of Niklaus Wirth's textbook "Algorithms and Data Structures" from 1985:

> *Procedure* `sort` *activates itself recursively. Such use of recursion in algorithms is a very powerful tool and will be discussed further in Chapter 3. In some programming languages of older provenience, recursion is disallowed for certain technical reasons. We will now show how this same algorithm can be expressed as a non-recursive procedure. Obviously, the solution is to express recursion as an iteration, whereby a certain amount of additional bookkeeping operations become necessary.*

Dijkstra's shunting-yard algorithm and Landin's SECD machine were directly written for programming languages of 'old provenience.' In Section 5 and in our earlier work [18, 24], we have shown how each is the defunctionalized and CPS-transformed counterpart of a recursive program in direct style. Incidentally, the same can be said of the quicksort algorithm in Wirth's book: CPS-transforming and defunctionalizing the recursive definition that precedes the paragraph above in the book yields the iterative definition that immediately follows in the book. So the 'certain amount of additional bookkeeping operations' mentioned above can be mechanized using the CPS transformation and defunctionalization. The same could be said for their respective correctness proofs: instead of developing them separately [30, 50], these proofs could be considered in the light of defunctionalization and refunctionalization [25, Section 5]—a future work.

Acknowledgments: The first author is grateful to Tarmo Uustalu for his invitation to MPC '06. Thanks are also due to Dariusz Biernacki and Kristian Støvring for comments. This work is partly supported by the Danish Natural Science Research Council, Grant no. 272-06-0530.

References

[1] Mads Sig Ager. *Partial Evaluation of String Matchers & Constructions of Abstract Machines*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS RS-04-3.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the technical report BRICS RS-04-28.

[5] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, number 2215 in Lecture Notes in Computer Science, pages 420–447, Sendai, Japan, October 2001. Springer-Verlag.

[6] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, June 1997. ACM Press.

[7] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[8] Dariusz Biernacki. *The Theory and Practice of Programming Languages with Delimited Continuations*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[9] Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. In Maurice Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium, LOPSTR 2003*, number 3018 in Lecture Notes in Computer Science, pages 143–159, Uppsala, Sweden, August 2003. Springer-Verlag.

[10] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274–297, 2006.

[11] Anders Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Rapport 90/17.

[12] Urban Boquist. *Code Optimization Techniques for Lazy Functional Languages*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Göteborg University, Göteborg, Sweden, April 1999.

[13] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[14] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 56–71, Berlin, Germany, March 2000. Springer-Verlag.

[15] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[16] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. *Higher-Order and Symbolic Computation*, 2007. To appear. A preliminary version was presented at the 1990 ACM Conference on Lisp and Functional Programming.

[17] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhaüser, 1993.

[18] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the technical report BRICS RS-03-33.

[19] Olivier Danvy. *An Analytical Approach to Program as Data Objects*. DSc thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2006.

[20] Olivier Danvy. Refunctionalization at work. In *Preliminary proceedings of the 8th International Conference on Mathematics of Program Construction (MPC '06)*, Kuressaare, Estonia, July 2006. Invited talk.

[21] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[22] Olivier Danvy and Mayer Goldberg. There and back again. *Fundamenta Informaticae*, 66(4):397–413, 2005. A preliminary version was presented at the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP 2002).

[23] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

[24] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number 4015 in Lecture Notes in Computer Science, pages 55–73, Dublin, Ireland, September 2005. Springer-Verlag. Extended version available as the technical report BRICS RS-06-17 (December 2006).

[25] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.

[26] Matthias Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

[27] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. `<http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html>`, 1989-2003.

[28] Jean-Christophe Filliâtre and François Pottier. Producing all ideals of a forest, functionally. *Journal of Functional Programming*, 13(5):945–956, 2003.

[29] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Simon Peyton Jones, editor, *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, SIGPLAN Notices, Vol. 37, No. 9, pages 235–246, Pittsburgh, Pennsylvania, September 2002. ACM Press.

[30] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–469, July 1999.

[31] Peter Henderson and James H. Morris Jr. A lazy evaluator. In Susan L. Graham, editor, *Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 95–103. ACM Press, January 1976.

[32] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[33] Werner E. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[34] Pierre Lescanne. From $\lambda\sigma$ to $\lambda\nu$ a journey through calculi of explicit substitutions. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 60–69, Portland, Oregon, January 1994. ACM Press.

[35] John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.

[36] Jan Midtgaard. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, June 2007. Forthcoming.

[37] Kevin Millikin. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, May 2007. Forthcoming.

[38] Lasse R. Nielsen. A denotational investigation of defunctionalization. Research Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000.

[39] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[40] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19(1):125–162, 2006. A preliminary version was presented at the Thirty-First Annual ACM Symposium on Principles of Programming Languages (POPL 2004).

[41] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [43].

[42] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[43] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, 1998.

[44] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[45] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[46] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[47] Andrew P. Tolmach and Dino P. Oliva. From ML to Ada: strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[48] Christopher P. Wadsworth. Some unusual λ-calculus numeral schemes. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 215–230. Academic Press, 1980.

[49] David H. D. Warren. Higher-order extensions to PROLOG: are they needed? In J. E. Hayes, Donald Michie, and Y.-H. Pao, editors, *Machine Intelligence*, volume 10, pages 441–454. Ellis Horwood, 1982.

[50] Kwangkeun Yi. "Proof-directed debugging" revisited for a first-order version. *Journal of Functional Programming*, 16(6):663–670, 2006.

Recent BRICS Report Series Publications