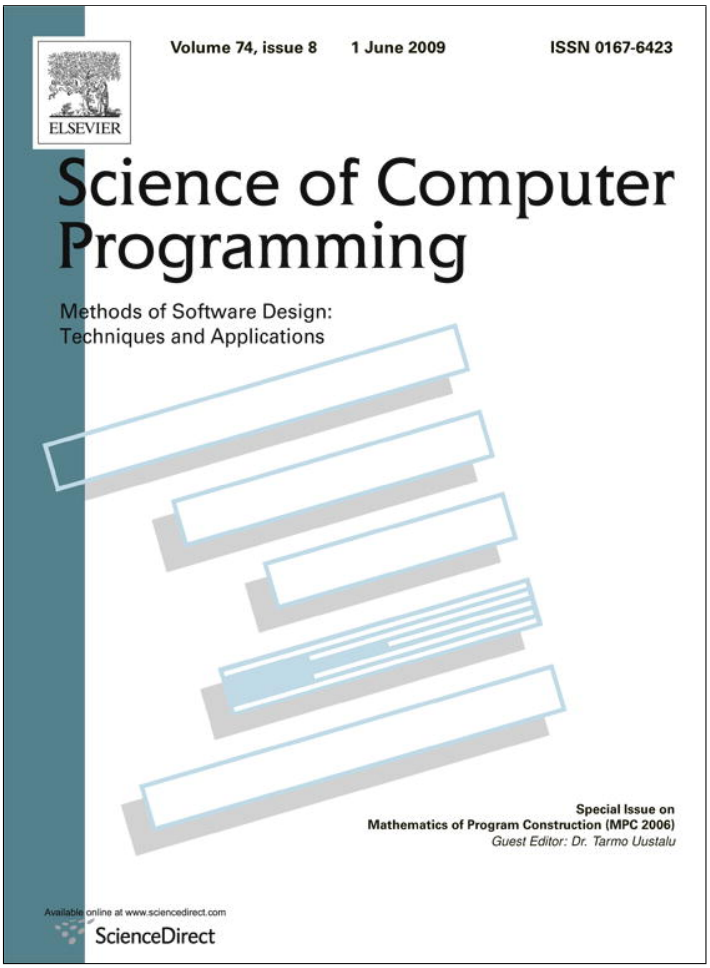


Science of Computer Programming 74 (2009) 534–549

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Refunctionalization at work

Olivier Danvy *, Kevin Millikin

Department of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark

ARTICLE INFO

Article history:
Received 21 March 2007
Received in revised form 3 August 2007
Accepted 15 October 2007
Available online 23 February 2009

Keywords:
Defunctionalization
Refunctionalization
Abstract machines
Continuations
Continuation-passing style (CPS)
Shunting-yard algorithm

ABSTRACT

We present the left inverse of Reynolds' defunctionalization and we show its relevance to programming and to programming languages. We propose two methods to transform 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.

1. Introduction

This article is a continuation of Danvy and Nielsen's earlier article "Defunctionalization at Work" [32], that outlined the extent to which Reynolds' defunctionalization [59] 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 *de*functionalization, 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 [59]. He presented it as a mere programming technique, and, except for deriving a first-order semantics in his textbook on programming languages [62, Section 12.4], he never used it again [61]. Since then, defunctionalization has been chiefly used in compilers [15,17], program transformers [68], and partial evaluators [14,20]. It has also been independently discovered in the context of logic programming [70], and subsequently it has been formalized in a typed setting [5,6,56,58]. Today it is being used for programming the web [44] and for modeling aspect-oriented programming [38].

Over the last few years [1,10,25,53,54], Danvy and his students have retraced Reynolds' 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 [2–4,7,11,24,31,63–65], including Felleisen et al.'s CEK machine, which was actually designed as such [41, page 196]. A practical byproduct of

* Corresponding author. Tel.: +45 89 42 93 39; fax: +45 89 42 56 01.
E-mail addresses: danvy@brics.dk (O. Danvy), kmillikin@brics.dk (K. Millikin).
URLs: http://www.brics.dk/~danvy/ (O. Danvy), http://www.brics.dk/~kmilli/ (K. Millikin).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.10.007

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 [25,54] that a number of them can be restated to be so. The goal of this article is to propose two techniques – disentangling and merging apply functions – to restate programs that are almost in defunctionalized form into ones that actually are in defunctionalized form (see Section 3). These programs can then be refunctionalized into a higher-order counterpart. In particular, it is our observation that refunctionalizing abstract machines always gives rise to a continuation-passing program.

Of course, not all defunctionalized programs and not all refunctionalized programs are interesting independently of each other, but still 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 [52] and, e.g., Henderson and Morris' iterative solution based on lazy lists [47] are the defunctionalized/refunctionalized counterparts of each other [13, Section 5]. Another example is the reverse function: Hughes' efficient reverse function based on curried list constructors [48] and the usual accumulator-based fast reverse function are the defunctionalized/refunctionalized counterparts of each other [32]. More examples are available elsewhere [25,32,42,53,54] 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 [60]. Yet his SECD machine is in defunctionalized form and the defunctionalized counterpart of a compositional continuation-passing evaluation function [24]. Furthermore, the *J* operator captures the current dump, i.e., in refunctionalized form, the continuation of the caller [31]. This observation has led us to suggest that Landin's name be added to the list of co-discoverers of continuations [31, Section 8].

1.3. Overview

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

Prerequisites. We expect a basic familiarity with Standard ML,[1] with continuation-passing style (CPS) [28,57,67], and with the left inverse of the CPS transformation [30]. The presentation of defunctionalization below should be self-contained, but the reader should not deny himself the pleasure of re-reading "Definitional Interpreters" [59].

2. Refunctionalization: The left inverse of defunctionalization

In his study of definitional interpreters for programming languages [59], 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

Operationally, 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 [5,6,32,56,58] considered all functions of a given type, or sets of functions determined by a control-flow analysis [66]. In any case, the set has to be closed 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.

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

```
signature ENVIRONMENT
= sig
    type ide
    type 'a env
    exception UNBOUND of ide
    val empty  : 'a env
    val extend : ide * 'a * 'a env -> 'a env
    val lookup : ide * 'a env -> 'a
  end
```

Fig. 1. An environment signature.

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 $\{x_j^i : \tau_j^i \mid 1 \leq j \leq m_i\}$, 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 one of the abstractions. There are thus n variants and variant i represents the values of the variables occurring free in abstraction i:

$$\begin{aligned} \mathit{datatype}\ \tau\_\mathit{arrow}\_\tau' = {} & C_1\ \mathit{of}\ \tau_1^1 \times \ldots \times \tau_{m_1}^1 \\ & \mid \ldots \\ & \mid C_n\ \mathit{of}\ \tau_1^n \times \ldots \times \tau_{m_n}^n \end{aligned}$$

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:

$$\begin{aligned} \mathit{fun}\ \mathit{apply}(f, x) = {} & \mathit{case}\ f\ \mathit{of} \\ & C_1(x_1^1, \ldots, x_{m_1}^1) \Rightarrow M_1 \\ & \mid \ldots \\ & \mid C_n(x_1^n, \ldots, x_{m_n}^n) \Rightarrow M_n \end{aligned}$$

The apply function may be curried or not; here, it is not.

(2) **Replace abstractions:** An abstraction $\lambda x.M_i$ with free variables $\{x_j^i : \tau_j^i \mid 1 \leq j \leq m_i\}$ is replaced with an application of the data-type constructor, $C_i(x_1^i, \ldots, x_{m_i}^i)$.

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, $\mathit{apply}(f, x)$.

2.2. Example: From higher-order to first-order environments

An environment is a mapping from identifiers to values that minimally satisfies the ML signature displayed in Fig. 1. To satisfy this signature, a module should define a representation of identifiers (e.g., as strings), a type of environments that is polymorphic with respect to the values denoted by the identifiers, a representation of an empty environment, a function `extend` to extend a given environment with a given identifier and a given value, a function `lookup` to retrieve the value associated with a given identifier in a given environment, and an exception `UNBOUND` to be raised by `lookup` if the given identifier is not in the domain of the given environment.

Environments are traditionally implemented either as functions from identifiers to values or as association lists. In this section we show that defunctionalizing the former implementation yields the latter one.

Here is a functional implementation:

```
structure Higher_Order_Environment :> ENVIRONMENT
= struct
    type ide = string
    type 'a env = ide -> 'a  (* the environment as a function *)
    exception UNBOUND of ide

    val empty = fn y => raise (UNBOUND y)

    fun extend (x, v, e) = fn y => if x = y then v else e y

    fun lookup (y, e) = e y
  end
```

[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.

Here, an environment is a function mapping an identifier to a value or raising the exception `UNBOUND`. Looking up an identifier in an environment is achieved by applying this environment to this identifier.

Let us defunctionalize this representation of environments. In Standard ML, the opaque signature ascription (`:>`) prevents this function type from being used outside of this structure, and therefore we can proceed independently of the use of this structure elsewhere in a program.

Only two abstractions give rise to inhabitants of the function type `ide -> 'a`:

- `fn y => raise (UNBOUND y)`,
  which has no free variables (we do not consider the exception name `UNBOUND` to be free because it will be lexically visible in the apply function), and
- `fn y => if x = y then v else e y`,
  which has three free variables: `x` of type `ide`, `v` of type `'a`, and `e` of type `ide -> 'a`.

Let us follow the steps outlined in Section 2.1:

(1) **Introduce a first-order data type:** The function type `ide -> 'a` is replaced by the following new data type:

```
datatype 'a ide_arrow_alpha = C1
                            | C2 of ide * 'a * 'a ide_arrow_alpha
```

This data type is isomorphic to `(ide * 'a) list`, which we use in the sequel.

Introduce an apply function: The apply function dispatches on the constructors of the first-order data type (`[]` and `::`), and interprets each of them as the body of the corresponding higher-order function:

```
fun apply ([], y)
    = raise UNBOUND y
  | apply ((x, v) :: e, y)
    = if x = y then v else e y
```

(2) **Replace abstractions:** The two abstractions are replaced with `[]` and `(x, v) :: e` respectively.

Replace applications: The two applications, `e x` in the original `lookup` function and `e y` in the apply function just above, are respectively replaced with `apply (e, x)` and `apply (e, y)`.

The result of defunctionalization is the traditional first-order representation of environments as an association list:

```
structure First_Order_Environment :> ENVIRONMENT
= struct
    type ide = string
    type 'a env = (ide * 'a) list  (* the environment as an association list *)
    exception UNBOUND of ide

    val empty = []

    fun extend (x, v, e) = (x, v) :: e

    fun apply ([], y)
        = raise (UNBOUND y)
      | apply ((x, v) :: e, y)
        = if x = y then v else apply (e, y)

    fun lookup (y, e) = apply (e, y)
  end
```

2.3. Refunctionalization

Refunctionalization is the left inverse of defunctionalization, mapping data types and apply functions in the image of defunctionalization back to higher-order functions. Danvy and Nielsen were the first to consider an inverse of defunctionalization [32].

A program with a first-order data structure and an apply function dispatching on it is in the image of Reynolds' defunctionalization algorithm if this apply function is the sole point of consumption of values of the data type. More generally, if there is only one 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, $f\ x$.
Replace data-type constructor applications: Data-type constructor applications are replaced with abstractions based on the apply function. We assume an apply function:

$$\begin{aligned} &\mathit{fun}\ apply(f, x) = \mathit{case}\ f\ \mathit{of} \\ &\qquad\qquad C_1(x_1^1, \ldots, x_{m_1}^1) \Rightarrow M_1 \\ &\qquad\quad \mid \ \ldots \\ &\qquad\quad \mid \ C_n(x_1^n, \ldots, x_{m_n}^n) \Rightarrow M_n \end{aligned}$$

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 $(\lambda x.M_i)\{v_1/x_1^i, \ldots, v_{m_i}/x_{m_i}^i\}$, 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:** Since the apply function is never called, its definition is no longer needed.
Remove the data-type definition: Since 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 [69] of a data type, i.e., the functional representation of a data type as its case-dispatch function.

2.4. Example: From first-order to higher-order environments

In this section, symmetrically to Section 2.2, we show how refunctionalizing a traditional first-order implementation of an environment as an association list yields a traditional higher-order implementation of an environment as a function.

Here is a traditional first-order implementation using an association list:

```
structure First_Order_Environment' :> ENVIRONMENT
= struct
    type ide = string
    type 'a env = (ide * 'a) list
    exception UNBOUND of ide

    val empty = []

    fun extend (x, v, e) = (x, v) :: e

    fun lookup (y, e)
        = let fun visit []
                  = raise (UNBOUND y)
                | visit ((x, v) :: e)
                  = if x = y
                    then v
                    else visit e
          in visit e
          end
end
```

Here, an environment is an association list, i.e., a list of pairs of identifiers and values. An empty environment is represented as the empty list and extending a given environment with a given identifier and a given value is achieved by pairing them and consing the pair on the given environment. Looking up an identifier in an environment is achieved by traversing the association list in search for the first matching identifier and raising the exception `UNBOUND` if none does.

Let us refunctionalize this representation of environments, which we can do locally because of the opaque signature ascription (`:>`) that isolates the definition of this structure from its uses. We identify the association list as a first-order data structure with a sole point of consumption, namely the `visit` function. To make `visit` fit the pattern that the apply function is explicitly applied to two arguments, we first apply the first step of lambda-lifting [34,49] to `visit`, i.e., we make it scope-insensitive by passing it explicitly its free variable `y`:

```
fun lookup (y, e)
    = let fun visit ([], y)
              = raise (UNBOUND y)
            | visit ((x, v) :: e, y)
              = if x = y
                then v
                else visit (e, y)
      in visit (e, y)
      end
```

(The second step of lambda-lifting would be to make the definition of `visit` float to the same lexical level as that of `lookup`, thereby obtaining recursive equations.)

With that, we follow the steps outlined in Section 2.3:

(1) **Replace applications of the apply function:** We replace calls to the apply function, `visit (e, y)`, with calls to the applied function, `e y`.
Replace data-type constructor applications: We replace data-type constructor applications with abstractions based on the apply function:
- `[]` is replaced by `fn y => raise (UNBOUND y)`, and
- `(x, v) :: e` is replaced by `fn y => if x = y then v else e y`.

(2) We then remove the definition of `visit`.

The result of refunctionalization is the traditional higher-order representation of environments as a function mapping identifiers to values.

3. Towards putting programs in defunctionalized 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 (this characterization is due to Danvy and Nielsen [32]). In this section, we present two simple transformations for programs that almost, but do not quite, have this form: 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 transformations 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 or if the single point of consumption is not in a separate function. This situation can occur, for example, when a program is defunctionalized and then the apply function is inlined.

'Disentangling' consists of abstracting each case expression dispatching on a candidate 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.

Disentangling was first used for the transition function of the SECD machine [24]. Disentangling the transition function of an abstract machine 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. We illustrate this technique in Fig. 3 of Section 4 for a push-down automaton recognizing Dyck words.

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 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 $apply_1 : \delta \times \tau_1 \to \tau'$ and $apply_2 : \delta \times \tau_2 \to \tau'$ can be merged into a single function $apply : \delta \times (\tau_1 + \tau_2) \to \tau'$ 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. (Dually, we can also merge the codomains of the apply functions.)

Merging apply functions was first used to refunctionalize Burge's version of the SECD machine with the *J* operator [31]. This technique works when the possible apply functions all have the same return types. In abstract machines this is usually the case because the transition functions are all tail-recursive and thus share the same answer type, as illustrated in Fig. 4 of Section 4 for a push-down automaton recognizing Dyck words.

4. A worked-out example: Recognizing Dyck words

Dyck words are well-balanced words 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]` represents 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 in the list so far:

```
(*  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
```

Fig. 2. A Dyck-words recognizer in Standard ML.

```
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
```

Fig. 3. Dyck-words recognizer: disentangled version.

```
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.

Fig. 2 displays an ML program implementing this transition system.

The goal of this section is to refunctionalize this Dyck-words recognizer with respect to the counter. Graphically, we proceed as follows:

Fig. 2 —dis-entangle→ Fig. 3 —merge, apply functions→ Fig. 4 —refunc-tionalize→ Fig. 5 —transform to direct style→ Fig. 6

In Fig. 2, 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 Fig. 3: `run` solely dispatches over the list of parentheses, and `run_nil` and `run_par` solely dispatch over the counter.

The transition system displayed in Fig. 3 operates in lockstep with the original transition system,[3] 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 Fig. 4. It is in defunctionalized form with respect to the data type `nat` and its apply function `run_aux`.

The refunctionalized counterpart of the merged program in Fig. 4 is displayed in Fig. 5. 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 displayed in Fig. 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` [30], though of course using an exception would do as well here.

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

```
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 : nat * word option -> bool  *)
          and run_aux (ZERO  , NONE   ) = true
            | run_aux (SUCC c, NONE   ) = false
            | run_aux (ZERO  , SOME ps) = false
            | run_aux (SUCC c, SOME ps) = run (ps, c)
      in run (ps, ZERO)
      end
```

Fig. 4. 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
```

Fig. 5. Dyck-words recognizer: refunctionalized version.

```
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)
```

Fig. 6. Dyck-words recognizer: refunctionalized version expressed in direct style.

The counter that was explicit in the original recognizer (Fig. 2), the disentangled one (Fig. 3), the merged one (Fig. 4), and the refunctionalized one (Fig. 5) is now implicit in the direct-style program (Fig. 6). 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 the 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 with operator precedence from infix form (as a stream of tokens: literals, operators, and parentheses) to postfix form (again, as a stream of tokens) or to an abstract-syntax tree. It is named for the railroad shunting yard because it uses a pair of tracks (stacks): one to store operand subtrees and one to store operators until all the operands are complete.

This bottom-up parser is attributed to Dijkstra,[4] and was developed for and used in one of the first Algol 60 compilers. It is still used today to process binary expressions in the GCC parser for C and for Objective-C, for example.

The goal of this section is to refunctionalize a shunting-yard parser with respect to its stack of operators. The version we consider here maps a stream of tokens (integers, addition operator, multiplication operator, left parenthesis, or right parenthesis) to an abstract-syntax tree:

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

4 "In the summer of 1959 I had discovered how to implement subroutines that could call themselves, by the end of the year I saw how to use a stack for the evaluation of expressions and how to translate expressions from the usual infix notation into 'reverse Polish' (only we did not call it that)." [37]

```
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 Figs. 7–9 to construct lists of expressions.

The algorithm is defined with an abstract machine implementing a push-down automaton with two stacks. This state-transition system operates iteratively over a stack of subtrees and a stack of operators (XADD and XMUL) and parenthetical separators (XPAR):

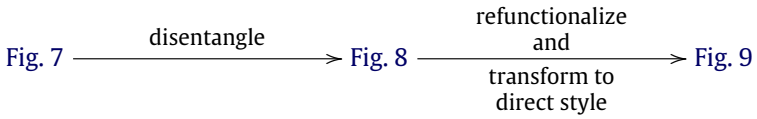
```
datatype operator = XADD | XMUL | XPAR
```

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.

Fig. 7 displays a program written in Standard ML that implements the shunting-yard algorithm. We proceed as follows to refunctionalize it with respect to the stack of operators:



As an ML program, the transition system in Fig. 7 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 Fig. 8: 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. This transition system operates in lockstep with the original one implemented in Fig. 7: the new machine takes one or two steps for each step in the original one.

In addition to being more readable, the transition system implemented in Fig. 8 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 Fig. 8 is in CPS. Furthermore – again save for errors in the input list of tokens – it uses its continuations linearly and in order, and can

```
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
```

Fig. 7. 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_nil (ts, [])
  end
```

Fig. 8. The shunting-yard algorithm: disentangled version.

```
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)
```

Fig. 9. The shunting-yard algorithm: refunctionalized version expressed in direct style.

therefore be expressed in direct style. We spare the reader with this CPS program (those who do not like these kinds of things etc.), and display the corresponding direct-style program in Fig. 9: 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` [30], though again using an exception would do as well.

The stack of operators that was explicit in Figs. 7 and 8 is now implicit in the direct-style program of Fig. 9. 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 still 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 with a reduce action.

6. Perspectives

Let us list other applications of refunctionalization (Section 6.1) and then mention its current limitations and alternatives (Section 6.2).

6.1. Other applications

We had to intervene in the following situations 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: the C and the D components can be refunctionalized into a control continuation and a dump continuation, respectively [24].

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 [16] and a version due to Felleisen [39]. Disentangling Felleisen's version yields a machine that is in defunctionalized form (and uses a control delimiter). In contrast, disentangling Landin and Burge's version is not sufficient to put it in defunctionalized form; its apply functions also need to be merged [31].

Properly tail-recursive stack inspection: In Clements and Felleisen's abstract machine for properly tail-recursive stack inspection [18], stack inspection is achieved by traversing the current context and checking its permission tables. Unzipping this context into an ordinary CEK-machine context and a list of permission tables yields a CEK machine with a state and an error facility: this machine can be refunctionalized and mapped back to direct style all the way to a compositional monadic evaluation function with a state+error monad [4]. Conversely, via refocusing [33], it syntactically corresponds to a new version of Fournet and Gordon's λ_{sec}-calculus [9, Section 7].

Full-reduction strategies: In our ongoing work on fully reducing abstract machines [54,55], we use both disentangling and merging to refunctionalize a number of abstract machines [22,21,45,50,51] into compositional normalization functions.

6.2. Limitations and alternatives

Dynamic CPS: The original abstract machine for dynamic delimited-continuation operators [40] is not in defunctionalized form. Disentangling it and then merging its apply functions was not enough for our purpose: we also needed to thread a tail of delimited continuations [12]. It was then simple to make it correspond, through refocusing, to a reduction semantics, and through refunctionalization, to a compositional evaluation function. Still, the trail of delimited continuations is an ad hoc device for which we have found no other use so far.

Towards deobjectification and reobjectification: Object orientation offers an alternative to merging apply functions with different domain and codomain types: the data type could be represented by an abstract class with a method for each of the apply functions. Each variant of the data type would be represented by a subclass of this abstract class with a field for each of the variant's free variables. This 'reobjectified' program can be implemented in a language without objects using any of the standard encodings of objects in objectless languages. The original 'deobjectified' program would then be such an encoding, specialized to flat class hierarchies.

7. Conclusion

We have investigated the applicability of refunctionalization for transforming programs and implementing programming languages. Elsewhere, we have illustrated its relevance to programming [13,29,32] and to specifying programming languages [2–4,11,24,31].

7.1. Wirth

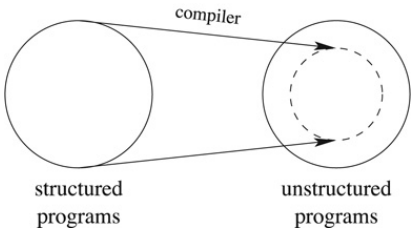
In some sense, our work on defunctionalization and refunctionalization provides a concrete illustration for the paragraph that follows the definition of the quicksort algorithm, in Section 2.3.3 of Niklaus Wirth's textbook "Algorithms and Data Structures" from 1985:

> *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 [24,31], 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 [46,71], these proofs could be considered in the light of defunctionalization and refunctionalization [32, Section 5]—a future work.

7.2. Dijkstra

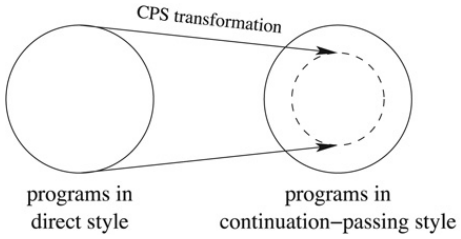
On the whole, Dijkstra's case against the GOTO statement [36] has been interpreted in the negative, as the forceful denunciation of a programming sin. It seems to us, though, that his message, 40 years on, can also be understood as an invitation to mindfulness when programming. Indeed consider a compiler from a structured language (e.g., one with while loops and conditional commands) to an unstructured language (e.g., one with labels and with conditional and unconditional jumps): on the one hand, this compiler yields programs that use GOTO statements; on the other hand, as denotations of structured programs, these unstructured programs only use GOTO statements to implement the control structures of structured programs. In that light, Dijkstra's implicit message is not so much that GOTO statements should be considered harmful, no matter what, than one should be mindful about staying in or straying from the image of the compiler when programming in the unstructured language:



In fact, finding oneself straying with good reason is a clear indication that a useful control construct is missing in the source language. For example, C and Pascal programmers condone the use of GOTO for error cases because these languages lack an exception mechanism.

Dijkstra's implicit message applies to at least two situations involving a program transformation and its left inverse:

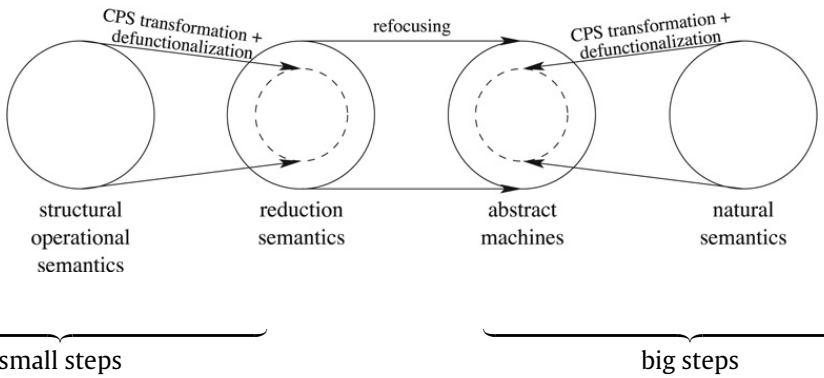
CPS transformation. When programming in continuation-passing style, one should be mindful of the continuation identifiers and of the parameters of continuations to stay in the image of the CPS transformation [23,26]:



Yet programmers use "the extra expressive power of CPS" to stray with good reason:

- for a simple example, not using the current continuation identifier prevents the computation from continuing and therefore has the effect of aborting it; this effect can be obtained in direct style by adding an "abort" control operator;
- for a common example, using another continuation identifier than the current one makes the computation continue elsewhere; this effect can be obtained in direct style by adding, e.g., the control operator escape [59] or call/cc [19,30];
- for a more radical example, one may altogether leave the language of CPS, where all calls are tail calls [67], and mix CPS with non-tail calls, thereby introducing the notion of delimited control [25, Section 1.3]; this effect can be obtained in direct style by adding, e.g., the delimited-control operators shift and reset [27].

Defunctionalization. Since, as pointed out in Section 1.1, defunctionalizing a CPS program yields an abstract machine, one should be mindful about specifying abstract machines in defunctionalized form:



Indeed

- CPS-transforming and defunctionalizing a function implementing a small-step semantics yields a function implementing a reduction semantics [25];
- symmetrically, CPS-transforming and defunctionalizing a function implementing a big-step semantics yields a function implementing an abstract machine [2–4,7,11,24,31]; and
- refocusing the evaluation function of a reduction semantics yields a function implementing an abstract machine [8,9,33].

Straying from the image of the CPS transformation makes it possible for the reduction semantics on the left and for the abstract machine on the right to specify, e.g., control effects.

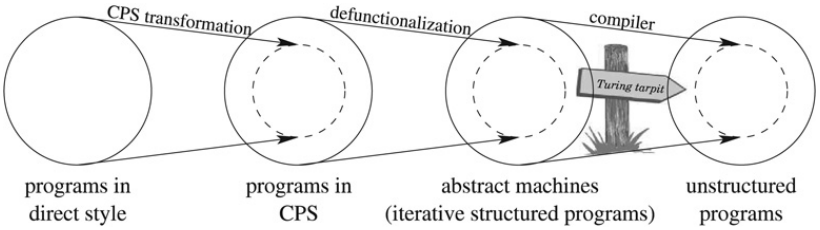
In any case, most abstract machines have been designed independently of defunctionalization. It is our experience that a number of them are in defunctionalized form, and that a number of others can be transformed to be so, using the techniques described here (see Section 6) or using, e.g., GADTs to make them well typed [58]. We currently have no sense about why one would want an abstract machine to stray from being in defunctionalized form when using a functional metalanguage for specifying computation.

So overall, we see the CPS transformation and defunctionalization as useful guidelines that are consistent with Dijkstra's implicit message.

Getting back to the shunting-yard algorithm, it seems very plausible that Dijkstra programmed it initially in the image of the above-mentioned compiler, while mentally picturing it as a structured program:

```
initialize the operator stack to be empty;
initialize the operand stack to be empty;
WHILE there are tokens in the input stream
DO peek the first such token;
   IF this first token is a number
   THEN read it and push it on the operand stack
   ELSIF this first token is an operator
   THEN IF the stack of operators is empty or has, at its top,
           a parenthetical separator or an operator with a lower precedence
        THEN read this first token and push it on the operator stack
        ELSE reduce
   ELSIF this first token is a left parenthesis
   THEN read it and push a parenthetical separator on the operator stack
   ELSIF this first token is a right parenthesis
   THEN IF the top operator is a parenthetical separator
        THEN read this first token and pop the top operator off the stack
        ELSE reduce
OD;
WHILE the operator stack is non-empty
DO reduce
OD
(* The result is on top of the operand stack. *)
```

In Section 5, we have (1) observed that the shunting-yard algorithm takes the form of an abstract machine, and (2) nudged it to be in defunctionalized form. Refunctionalizing it yields a program in CPS. Mapping this program back to direct style yields a functional parser, farther and farther away from the Turing tarpit:



To close, when Dijkstra submitted "A Case against the GO TO Statement" [35] to CACM 40 years ago, Wirth used his editorial discretion to change the title to "Go To Statement Considered Harmful" [36]. All proportions kept, we would be grateful to our editor if he could leave the title of the present article as it is now: witness the drawing just above, abstract machines in defunctionalized form are harmless. As for those that can be disentangled etc., it is our point that they are mostly so.

Acknowledgements

The first author is grateful to Tarmo Uustalu for his invitation to MPC '06, and doubly so for not changing our title.[5] Thanks are also due to Dariusz Biernacki, Kristian Støvring, and the anonymous reviewers of *Science of Computer Programming* for

5 But just in case, it should be "Refunctionalization at Work."

their comments, and to Oege de Moor for directing us to Floyd's parser [43], which we have found to be similar to Dijkstra's shunting-yard algorithm, but the two stacks zipped together. This work is partly supported by the Danish Natural Science Research Council, Grant no. 272-06-0530.

References

[1] M.S. Ager, Partial evaluation of string matchers & constructions of abstract machines, Ph.D. Thesis, BRICS, University of Aarhus, 2006.
[2] M.S. Ager, D. Biernacki, O. Danvy, J. Midtgaard, A functional correspondence between evaluators and abstract machines, in: Proc. of 5th Int. ACM-SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP 2003, Uppsala, Aug. 2003, ACM Press, New York, 2003, pp. 8–19.
[3] M.S. Ager, O. Danvy, J. Midtgaard, A functional correspondence between call-by-n eed evaluators and lazy abstract machines, Inform. Process. Lett. 90 (5) (2004) 223–232. Extended version available as Research Report BRICS RS-04-3.
[4] M.S. Ager, O. Danvy, J. Midtgaard, A functional correspondence between monadic evaluators and abstract machines for languages with computational effects, Theoret. Comput. Sci. 342 (1) (2005) 149–172. Extended version available as Research Report BRICS RS-04-28.
[5] A. Banerjee, N. Heintze, J.G. Riecke, Design and correctness of program transformations based on control-flow analysis, in: N. Kobayashi, B.C. Pierce (Eds.), Proc. of 4th Int. Symp. on Theoretical Aspects of Computer Software, TACS 2001, Sendai, Oct. 2001, in: Lect. Notes in Comput. Sci., vol. 2215, Springer, Berlin, 2001, pp. 420–447.
[6] J.M. Bell, F. Bellegarde, J. Hook, Type-driven defunctionalization, in: Proc. of 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP '97, Amsterdam, June 1997, ACM Press, New York, 1997, pp. 25–37. (= ACM SIGPLAN Notices 32 (8)).
[7] M. Biernacka, D. Biernacki, O. Danvy, An operational foundation for delimited continuations in the CPS hierarchy, Logical Methods Comput. Sci. 1 (2) (2005) article 5.
[8] M. Biernacka, O. Danvy, A concrete framework for environment machines, ACM Trans. Comput. Logic 9 (1) (2007) article 6.
[9] M. Biernacka, O. Danvy, A syntactic correspondence between context-sensitive calculi and abstract machines, Theoret. Comput. Sci. 375 (1–3) (2007) 76–108. Extended version available as Research Report BRICS RS-06-18.
[10] D. Biernacki, The theory and practice of programming languages with delimited continuations, Ph.D. Thesis, BRICS, University of Aarhus, 2005.
[11] D. Biernacki, O. Danvy, From interpreter to logic engine by defunctionalization, in: M. Bruynooghe (Ed.), Revised Selected Papers from 13th Int. Symp. on Logic-Based Program Synthesis and Transformation, LOPSTR 2003, Uppsala, Aug. 2003, in: Lect. Notes in Comput. Sci., vol. 3018, Springer, Berlin, 2004, pp. 143–159.
[12] D. Biernacki, O. Danvy, K. Millikin, A dynamic continuation-passing style for dynamic delimited continuations, ACM Trans. Program. Lang. Syst. (accepted).
[13] D. Biernacki, O. Danvy, C. Shan, On the static and dynamic extents of delimited continuations, Sci. Comput. Program. 60 (3) (2006) 274–297.
[14] A. Bondorf, Self-applicable partial evaluation, Ph.D. Thesis, DIKU Rapport 90/17, Computer Science Dept. University of Copenhagen, 1990.
[15] U. Boquist, Code optimization techniques for lazy functional languages, Ph.D. Thesis, Dept. of Computing Science, Chalmers University of Technology and Göteborg University, 1999.
[16] W.H. Burge, Recursive Programming Techniques, Addison-Wesley, Reading, MA, 1975.
[17] H. Cejtin, S. Jagannathan, S. Weeks, Flow-directed closure conversion for typed languages, in: G. Smolka (Ed.), Proc. of 9th Europ. Symp. on Programming, ESOP 2000, Berlin, March/Apr. 2000, in: Lect. Notes in Comput. Sci., vol. 1782, Springer, Berlin, 2000, pp. 56–71.
[18] J. Clements, M. Felleisen, A tail-recursive semantics for stack inspection, ACM Trans. Program. Lang. Syst. 26 (6) (2004) 1029–1052.
[19] W. Clinger, D.P. Friedman, M. Wand, A scheme for a higher-level semantic algebra, in: M. Nivat, J.C. Reynolds (Eds.), Algebraic Methods in Semantics, Cambridge University Press, Cambridge, 1985, pp. 237–250.
[20] C. Consel, A tour of Schism: A partial evaluation system for higher-order applicative languages, in: Proc. of 1993 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '93, Copenhagen, June 1993, ACM Press, New York, 1993, pp. 145–154.
[21] P. Crégut, Strongly reducing variants of the Krivine abstract machine, Higher-Order Symb. Comput. 20 (3) (2007) 209–230. Revision of a paper that originally appeared in Proc. of 1990 ACM Conf. on Lisp and Functional Programming, LFP '90 (Nice, June 1990) (ACM Press, New York, 1990) 333–340.
[22] P.-L. Curien, Categorical Combinators, Sequential Algorithms and Functional Programming, 2nd ed., in: Progress in Theoretical Computer Science, Birkhäuser, Boston, 1993.
[23] O. Danvy, Back to direct style, Sci. Comput. Program. 22 (3) (1994) 183–195.
[24] O. Danvy, A rational deconstruction of Landin's SECD machine, in: C. Grelck, F. Huch, G.J. Michaelson, P. Trinder (Eds.), Revised Selected Papers from 16th Int. Wksh. on Implementation and Application of Functional Languages, IFL '04, Lübeck, Sept. 2004, in: Lect. Notes in Comput. Sci., vol. 3474, Springer, Berlin, 2005, pp. 52–71. Extended version available as Research Report BRICS RS-03-33.
[25] O. Danvy, An analytical approach to program as data objects, DSc Thesis, Dept. of Computer Science, University of Aarhus, 2006.
[26] O. Danvy, B. Dzafic, F. Pfenning, On proving syntactic properties of CPS programs, in: A. Gordon, A. Pitts (Eds.), Proc. of 3rd Int. Wksh. on Higher-Order Operational Techniques in Semantics, HOOTS '99, Paris, Sept./Oct. 1999, in: Electron. Notes in Theor. Comput. Sci., vol. 26, Elsevier, Amsterdam, 1999, pp. 21–33.
[27] O. Danvy, A. Filinski, Abstracting control, in: Proc. of 1990 ACM Conf. on Lisp and Functional Programming, LFP '90, Nice, June 1990, ACM Press, New York, 1990, pp. 151–160.
[28] O. Danvy, A. Filinski, Representing control, a study of the CPS transformation, Math. Struct. Comput. Sci. 2 (4) (1992) 361–391.
[29] O. Danvy, M. Goldberg, There and back again, Fund. Inform. 66 (4) (2005) 397–413.
[30] O. Danvy, J.L. Lawall, Back to direct style II: First-class continuations, in: Proc. of 1992 ACM Conf. on Lisp and Functional Programming, San Francisco, CA, June 1992, ACM Press, New York, 1992, pp. 299–310. (= LISP Pointers 5(1)).
[31] O. Danvy, K. Millikin, A rational deconstruction of Landin's J operator, Logical Methods Comput. Sci. 4 (4) (2008) article 12.
[32] O. Danvy, L.R. Nielsen, Defunctionalization at work, in: Proc. of 3rd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'01, Florence, Sept. 2001, ACM Press, New York, 2001, pp. 162–174. Extended version available as Research Report BRICS RS-01-23.
[33] O. Danvy, L.R. Nielsen, Refocusing in reduction semantics, research report BRICS RS-04-26, DAIMI, Dept. of Computer Science, University of Aarhus, 2004, in: Prel. version in Proc. of 2nd Int. Wksh. on Rule-Based Programming, RULE 2001, Electron. Notes in Theor. Comput. Sci. 59 (4) (2001) 358–374.
[34] O. Danvy, U.P. Schultz, Lambda-lifting in quadratic time, J. Funct. Logic Program (2004) article 1.
[35] E.W. Dijkstra, A case against the GO TO statement, EWD 215, 1968, Available online at: http://www.cs.utexas.edu/users/EWD/.
[36] E.W. Dijkstra, Go To statement considered harmful, Commun. ACM 11 (3) (1968) 147–148.
[37] E.W. Dijkstra, From my life, EWD 1166, 1993, Available online at: http://www.cs.utexas.edu/users/EWD/.
[38] C.J. Dutchyn, Specializing continuations: A model for dynamic join points, in: Proc. of 6th Wksh. on Foundations of Aspect-Oriented Languages, FOAL '07, Vancouver, BC, March 2007, in: ACM Int. Conf. Proc. Series, vol. 206, ACM Press, New York, 2007, pp. 45–57.
[39] M. Felleisen, Reflections on Landin's J operator: A partly historical note, Comput. Lang. 12 (3–4) (1987) 197–207.
[40] M. Felleisen, The theory and practice of first-class prompts, in: Conf. Record of 15th Ann. ACM Symposium on Principles of Programming Languages, POPL '88, San Diego, CA, Jan. 1988, ACM Press, New York, 1988, pp. 180–190.
[41] M. Felleisen, D.P. Friedman, Control operators, the SECD machine, and the λ-calculus, in: M. Wirsing (Ed.), Formal Description of Programming Concepts III, North-Holland, Amsterdam, 1987, pp. 193–217.
[42] J.-C. Filliâtre, F. Pottier, Producing all ideals of a forest, functionally, J. Funct. Programming 13 (5) (2003) 945–956.
[43] R.W. Floyd, Syntactic analysis and operator precedence, J. ACM 10 (3) (1963) 316–333.
[44] P.T. Graunke, R.B. Findler, S. Krishnamurthi, M. Felleisen, Automatically restructuring programs for the web, Autom. Softw. Eng. 11 (4) (2004) 337–364.

[45] B. Grégoire, X. Leroy, A compiled implementation of strong reduction, in: Proc. of 2002 ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2002, Pittsburgh, PA, Oct. 2002, ACM Press, New York, 2002, pp. 235–246. (= ACM SIGPLAN Notices 37 (2)).
[46] R. Harper, Proof-directed debugging, J. Funct. Programming 9 (4) (1999) 463–469.
[47] P. Henderson, J.H. Morris Jr., A lazy evaluator, in: Conf. Record of 3rd Ann. ACM Symposium on Principles of Programming Languages, POPL'76, Atlanta, GA, Jan. 1976, ACM Press, New York, 1976, pp. 95–103.
[48] J. Hughes, A novel representation of lists and its application to the function reverse, Inform. Process. Lett. 22 (3) (1986) 141–144.
[49] T. Johnsson, Lambda lifting: Transforming programs to recursive equations, in: J.-P. Jouannaud (Ed.), Proc. of 2nd ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'85, Nancy, Sept. 1985, in: Lect. Notes in Comput. Sci., vol. 201, Springer, Berlin, 1985, pp. 190–203.
[50] W.E. Kluge, Abstract Computing Machines: A Lambda Calculus Perspective, in: Texts in Theoretical Computer Science : An EATCS Series, Springer, Berlin, 2005.
[51] P. Lescanne, From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions, in: Conf. Record of 21st Ann. ACM Symp. on Principles of Programming Languages, POPL '94, Portland, OR, Jan. 1994, ACM Press, New York, 1994, pp. 60–69.
[52] J. McCarthy, Another samefringe, SIGART Newsletter 61 (1977).
[53] J. Midtgaard, Transformation, analysis, and interpretation of higher-order procedural programs, Ph.D. thesis, BRICS, University of Aarhus, 2007.
[54] K. Millikin, A structured approach to the transformation, normalization and execution of computer programs, Ph.D. Thesis, BRICS, University of Aarhus, 2007.
[55] J. Munk, A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state, Master's Thesis, Dept. of Computer Science, University of Aarhus, 2007.
[56] L.R. Nielsen, A denotational investigation of defunctionalization, Research Report BRICS RS-00-47, Dept. of Computer Science, University of Aarhus, 2000.
[57] G.D. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoret. Comput. Sci. 1 (1975) 125–159.
[58] F. Pottier, N. Gauthier, Polymorphic typed defunctionalization and concretization, Higher-Order Symb. Comput. 19 (1) (2006) 125–162.
[59] J.C. Reynolds, Definitional interpreters for higher-order programming languages, in: Proc. of 25th ACM Nat. Conf., Boston, MA, Aug. 1972, ACM Press, New York, 1972, pp. 717–740. Reprinted in Higher-Order and Symb. Comput. 11 (4) (1998) 363–397, with a foreword [61].
[60] J.C. Reynolds, The discoveries of continuations, Lisp Symb. Comput. 6 (3–4) (1993) 233–247.
[61] J.C. Reynolds, Definitional interpreters revisited, Higher-Order Symb. Comput. 11 (4) (1998) 355–361.
[62] J.C. Reynolds, Theories of Programming Languages, Cambridge University Press, Cambridge, 1998.
[63] D.A. Schmidt, State transition machines for lambda calculus expressions, in: N.D. Jones (Ed.), Proc. of Wksh. on Semantics-Directed Compiler Generation, Aarhus, Jan. 1980, in: Lect. Notes in Comput. Sci., vol. 94, Springer, Berlin, 1980, pp. 415–400.
[64] D.A. Schmidt, State-transition machines for lambda-calculus expressions, Higher-Order Symb. Comput. 20 (3) (2007) 319–332. Journal version of [63], with an afterword [65].
[65] D.A. Schmidt, State-transition machines, revisited, Higher-Order Symb. Comput. 20 (3) (2007) 333–335.
[66] O. Shivers, Control-flow analysis of higher-order languages or taming lambda, Ph.D. Thesis, Techn. Report CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.
[67] G.L. Steele Jr., Rabbit: A compiler for Scheme, Master's Thesis, Techn. Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1978.
[68] A.P. Tolmach, D.P. Oliva, From ML to Ada: Strongly-typed language interoperability via source translation, J. Funct. Programming 8 (4) (1998) 367–412.
[69] C.P. Wadsworth, Some unusual λ-calculus numeral schemes, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London, 1980, pp. 215–230.
[70] D.H.D. Warren, Higher-order extensions to PROLOG: Are they needed?, in: J.E. Hayes, D. Michie, Y.-H. Pao (Eds.), in: Machine Intelligence, vol. 10, Ellis Horwood, Chichester, 1982, pp. 441–454.
[71] K. Yi, "Proof-directed debugging" revisited for a first-order version, J. Funct. Programming 16 (6) (2006) 663–670.