Defunctionalization at Work *

Olivier Danvy and Lasse R. Nielsen
BRICS †
Department of Computer Science
University of Aarhus ‡

Abstract

Reynolds's defunctionalization technique is a whole-program transformation from higher-order to first-order functional programs. We study practical applications of this transformation and uncover new connections between seemingly unrelated higher-order and first-order specifications and between their correctness proofs. Defunctionalization therefore appears both as a springboard for revealing new connections and as a bridge for transferring existing results between the first-order world and the higher-order world.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (functional) programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, Control structures, Data types and structures, Procedures, functions, and subroutines, Recursion*; E.1 [**Data**]: Data Structures—*Records*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Functional constructs*; F.4.1 [**Mathematical Logic and Formal Languages**]: Lambda calculus and related systems; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program transformation.*

Keywords

Church encoding, closure conversion, continuations, continuation-passing style (CPS), CPS transformation, defunctionalization, direct-style transformation, first-order programs, higher-order programs, lambda-lifting, ML, regular expressions, Scheme, superconbinator conversion, syntactic theories.

‡Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
E-mail: {danvy,lrn}@brics.dk

*An extended version of this article is available as the BRICS technical report RS-01-23.

†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

PPDP 01 Florence, Italy

1. Background and Introduction

In first-order programs, all functions are named and each call refers to the callee by its name. In higher-order programs, functions may be anonymous, passed as arguments, and returned as results. As Strachey put it [49], functions are *second-class* denotable values in a first-order program, and *first-class* expressible values in a higher-order program. One may then wonder how first-class functions are represented at run time.

- First-class functions are often represented with *closures*, i.e., expressible values pairing a code pointer and the denotable values of the variables occurring free in that code, as proposed by Landin in the mid-1960's [30]. Today, closures are the most common representation of first-class functions in the world of eager functional programming [1, 17, 32], as well as a standard representation for implementing object-oriented programs [23]. They are also used to implement higher-order logic programming [8].

- Alternatively, higher-order programs can be *defunctionalized* into first-order programs, as proposed by Reynolds in the early 1970's [43]. In a defunctionalized program, first-class functions are represented with first-order data types: a first-class function is introduced with a constructor holding the values of the free variables of a function abstraction, and it is eliminated with a case expression dispatching over the corresponding constructors.

- First-class functions can also be dealt with by translating functional programs into *combinators* and using graph reduction, as proposed by Turner in the mid-1970's [51]. This implementation technique has been investigated extensively in the world of lazy functional programming [27, 29, 37, 38].

Compared to closure conversion and to combinator conversion, defunctionalization has been used very little. The goal of this article is to study practical applications of it.

We first illustrate defunctionalization with two concrete examples (Sections 1.1 and 1.2). In the first program, two function abstractions are instantiated once, and in the second program, one function abstraction is instantiated repeatedly. We then characterize defunctionalization in a nutshell (Section 1.3) before reviewing related work (Section 1.4). Finally, we raise questions to which defunctionalization provides answers (Section 1.5).

1.1 A sample higher-order program with a static number of closures

In the following ML program, `aux` is passed a first-class function, applies it to 1 and 10, and sums the results. The `main` function calls `aux` twice and multiplies the results. All in all, two function abstractions occur in this program, in `main`.

```
(*  aux : (int -> int) -> int  *)
fun aux f
    = f 1 + f 10

(*  main : int * int * bool -> int  *)
fun main (x, y, b)
    = aux (fn z => x + z) *
      aux (fn z => if b then y + z else y - z)
```

Defunctionalizing this program amounts to defining a data type with two constructors, one for each function abstraction, and its associated apply function. The first function abstraction contains one free variable (`x`, of type integer), and therefore the first data-type constructor requires an integer. The second function abstraction contains two free variables (`y`, of type integer, and `b`, of type boolean), and therefore the second data-type constructor requires an integer and a boolean.

In `main`, the first first-class function is thus introduced with the first constructor and the value of `x`, and the second with the second constructor and the values of `y` and `b`.

In `aux`, the functional argument is passed to a second-class function `apply` that eliminates it with a case expression dispatching over the two constructors.

```
datatype lam = LAM1 of int
             | LAM2 of int * bool

(*  apply : lam * int -> int  *)
fun apply (LAM1 x, z)
    = x + z
  | apply (LAM2 (y, b), z)
    = if b then y + z else y - z

(*  aux_def : lam -> int  *)
fun aux_def f
    = apply (f, 1) + apply (f, 10)

(*  main_def : int * int * bool -> int  *)
fun main_def (x, y, b)
    = aux_def (LAM1 x) * aux_def (LAM2 (y, b))
```

1.2 A sample higher-order program with a dynamic number of closures

A reviewer wondered what happens for programs that "dynamically generate" new closures, and whether such programs lead to new constants and thus require extensible case expressions. The following example illustrates such a situation and shows that no new constants and no extensible case expressions are needed.

In the following ML program, `aux` is passed two arguments and applies one to the other. The `main` function is given a number `i` and a list of numbers `[j1, j2, ...]` and returns the list of numbers `[i+j1, i+j2, ...]`. One function abstraction, `fn i => i + j`, occurs in this program, in `main`, as the second argument of `aux`. Given an input list of length n, the function abstraction is instantiated n times in the course of the computation.

```
(*  aux : int * (int -> int) -> int  *)
fun aux (i, f)
    = f i

(*  main = fn : int * int list -> int list  *)
fun main (i, js)
    = let fun walk nil
              = nil
            | walk (j :: js)
              = (aux (i, fn i => i + j)) :: (walk js)
      in walk js
      end
```

Defunctionalizing this program amounts to defining a data type with only one constructor, since there is only one function abstraction, and its associated apply function. The function abstraction contains one free variable (`j`, of type integer), and therefore the data-type constructor requires an integer.

In `main`, the first-class function is introduced with the constructor and the value of `j`.

In `aux`, the functional argument is passed to a second-class function `apply` that eliminates it with a case expression dispatching over the constructor.

```
datatype lam = LAM of int

(*  apply : lam * int -> int  *)
fun apply (LAM j, i)
    = i + j

(*  aux_def : int * lam -> int  *)
fun aux_def (i, f)
    = apply (f, i)

(*  main_def : int * int list -> int list  *)
fun main_def (i, js)
    = let fun walk nil
              = nil
            | walk (j :: js)
              = (aux_def (i, LAM j)) :: (walk js)
      in walk js
      end
```

Given an input list of length n, the constructor `LAM` is used n times in the course of the computation.

1.3 Defunctionalization in a nutshell

In a higher-order program, first-class functions arise as instances of function abstractions. All these function abstractions can be enumerated in a whole program. Defunctionalization is thus a whole-program transformation where function types are replaced by an enumeration of the function abstractions in this program.

Defunctionalization therefore takes its roots in type theory. Indeed, a function type hides typing assumptions from the context, and, as pointed out by Minamide, Morrisett, and Harper in their work on typed closure conversion [32], making these assumptions explicit requires an existential type. For a whole program, this existential type can be represented with a finite sum together with the corresponding injections and case dispatch, and this representation is precisely what defunctionalization achieves.

These type-theoretical roots do not make defunctionalization a straitjacket, to paraphrase Reynolds about Algol [42]. For example, one can use several apply functions, e.g., grouped by types, as in Bell, Bellegarde, and Hook's work [4]. One can also defunctionalize a program selectively, e.g., only its continuations, as in Section 3. One can even envision a lightweight defunctionalization similar to Steckler and Wand's lightweight closure conversion [47], as in Banerjee, Heintze, and Riecke's recent work [2].

1.4 Related work

Originally, Reynolds devised defunctionalization to transform a higher-order interpreter into a first-order one [43]. He presented it as a programming technique, and never used it again [44].

Since then, defunctionalization has not been used much, though when it has, it was as a full-fledged implementation technique: Bondorf uses it to make higher-order programs amenable to first-order partial evaluation [5]; Tolmach and Oliva use it to compile ML programs into Ada [50]; Fegaras uses it in his object-oriented database management system, lambda-DB [18]; Wang and Appel use it in type-safe garbage collectors [54]; and defunctionalization is an integral part of MLton [7] and of Boquist's Haskell compiler [6].

Only lately has defunctionalization been formalized: Bell, Bellegarde, and Hook showed that it preserves types [4]; Nielsen proved its partial correctness using denotational semantics [35, 36]; and Banerjee, Heintze and Riecke proved its total correctness using operational semantics [2].

1.5 This work

Functional programming encourages fold-like recursive descents, typically using auxiliary recursive functions. Often, these auxiliary functions are higher order in that their co-domain is a function space. For example, if an auxiliary function has an accumulator of type α, its co-domain is $\alpha \to \beta$, for some β. For another example, if an auxiliary function has a continuation of type $\alpha \to \beta$, for some β, its co-domain is $(\alpha \to \beta) \to \beta$. How do these functional programs compare to programs written using a first-order, data-structure oriented approach?

Wand's classical work on continuation-based program transformations [53] was motivated by the question "What is a data-structure continuation?". Each of the examples considered in Wand's paper required a eureka step to design a data structure for representing a continuation. Are such eureka steps always necessary?

Continuations are variously presented as a functional representation of the rest of the computation and as a functional representation of the context of a computation [20]. Wand's work addressed the former view, so let us consider the latter one. For example, in his PhD thesis [19], Felleisen developed a syntactic approach to semantics relying on the first-order notions of 'evaluation context' and of 'plugging expressions into contexts'. How do these first-order notions compare to the notion of continuation?

In the rest of this article, we show that defunctionalization provides a single answer to all these questions. All the programs we consider perform a recursive descent and use an auxiliary function. When this auxiliary function is higher-order, defunctionalization yields a first-order version with an accumulator (e.g., tree flattening in Section 2.1 and list reversal in Section 2.2). When this auxiliary function is first-order, we transform it into continuation-passing style; defunctionalization then yields an iterative first-order version with an accumulator in the form of a data structure (e.g., string parsing in Section 3.1 and regular-expression matching in Section 5). We also consider a definitional interpreter for a syntactic theory and we identify that it is written in a defunctionalized form. We then "refunctionalize" it and obtain a continuation-passing definitional interpreter whose continuations represents the evaluation contexts of the syntactic theory (Section 4).

In addition, we observe that defunctionalization and Church encoding have dual purposes, since Church encoding is a classical way to represent data structures with higher-order functions. What is the result of defunctionalizing a Church-encoded data structure? And what does one obtain when Church-encoding the result of defunctionalization?

Similarly, we observe that backtracking is variously implemented in a first-order setting with one or two stacks, and in a higher-order setting with one or two continuations. It is natural enough to wonder what is the result of Church-encoding the stacks and of defunctionalizing the continuations. One can wonder as well about the correctness proofs of these programs—how do they compare?

In the rest of this article, we also answer these questions. We defunctionalize two programs using Hughes's higher-order representation of intermediate lists and obtain two efficient and traditional first-order programs (Section 2.2). We also clarify the extent to which Church encoding and defunctionalization can be considered as inverses of each other (Sections 2.3, 2.4, and 2.5). Finally, we compare and contrast a regular-expression matcher and its proof before and after defunctionalization (Section 5).

2. Defunctionalization of List- and of Tree-Processing Programs

We consider several canonical higher-order programs over lists and trees and we defunctionalize them. In each case, defunctionalization yields a known, but unrelated solution. We then turn to Church encoding, which provides a uniform higher-order representation of data structures. We consider the result of defunctionalizing Church-encoded data structures, and we consider the result of Church-encoding the result of defunctionalization.

2.1 Flattening a binary tree into a list

To flatten a binary tree into a list of its leaves, we choose to map a leaf to a curried list constructor and a node to function composition, homomorphically. In other words, we map a list into the monoid of functions from lists to lists. This definition hinges on the built-in associativity of function composition.

```
datatype 'a bt = LEAF of 'a
               | NODE of 'a bt * 'a bt

(*  cons : 'a -> 'a list -> 'a list  *)
fun cons x xs
    = x :: xs

(*  flatten : 'a bt -> 'a list             *)
(*     walk : 'a bt -> 'a list -> 'a list  *)
fun flatten t
    = let fun walk (LEAF x)
              = cons x
            | walk (NODE (t1, t2))
              = (walk t1) o (walk t2)
      in walk t nil
      end
```

Eta-expanding the result of `walk` and inlining `cons` and `o` yields a curried version of the fast flatten function with an accumulator.

```
(*  flatten_ee : 'a bt -> 'a list             *)
(*        walk : 'a bt -> 'a list -> 'a list  *)
```

```
fun flatten_ee t
    = let fun walk (LEAF x) a
                = x :: a
            | walk (NODE (t1, t2)) a
                = walk t1 (walk t2 a)
      in walk t nil
      end
```

It is also instructive to defunctionalize `flatten`. Two functional values occur—one for the leaves and one for the nodes—and therefore they give rise to a data type with two constructors. Since `flatten` is homomorphic, the new data type is isomorphic to the data type of binary trees, and therefore the associated apply function could be made to work directly on the input tree, e.g., using deforestation [52]. At any rate, we recognize this apply function as an uncurried version of the fast flatten function with an accumulator.

```
datatype 'a lam = LAM1 of 'a
                | LAM2 of 'a lam * 'a lam

(*  apply : 'a lam * 'a list -> 'a list  *)
fun apply (LAM1 x, xs)
    = x :: xs
  | apply (LAM2 (f1, f2), xs)
    = apply (f1, apply (f2, xs))

(*  cons_def : 'a -> 'a lam  *)
fun cons_def x
    = LAM1 x

(*  o_def : 'a lam * 'a lam -> 'a lam  *)
fun o_def (f1, f2)
    = LAM2 (f1, f2)

(*  flatten_def : 'a bt -> 'a list  *)
(*         walk : 'a bt -> 'a lam   *)
fun flatten_def t
    = let fun walk (LEAF x)
                = cons_def x
            | walk (NODE (t1, t2))
                = o_def (walk t1, walk t2)
      in apply (walk t, nil)
      end
```

The monoid of functions from lists to lists corresponds to Hughes's novel representations of lists [28], which we treat next.

2.2 Higher-order representations of lists

In the mid-1980's, Hughes proposed to represent intermediate lists as partially applied concatenation functions [28], so that instead of constructing a list `xs`, one instantiates the function abstraction `fn ys => xs @ ys`. The key property of this higher-order representation is that lists can be concatenated in constant time. Therefore, the following naive version of `reverse` operates in linear time instead of in quadratic time, as with the usual linked representation of lists, where lists are concatenated in linear time.

```
(*  append : 'a list -> 'a list -> 'a list  *)
fun append xs ys
    = xs @ ys

(*  reverse : 'a list -> 'a list  *)
fun reverse xs
    = let fun walk nil
                = append nil
            | walk (x :: xs)
                = (walk xs) o (append [x])
      in walk xs nil
      end
```

Let us defunctionalize this program. First, like Hughes, we recognize that appending the empty list is the identity function and that appending a single element amounts to consing it.

```
(*  id : 'a list -> 'a list  *)
fun id ys
    = ys

(*  cons : 'a -> 'a list -> 'a list *)
fun cons x xs
    = x :: xs

(* reverse : 'a list -> 'a list             *)
(*    walk : 'a list -> 'a list -> 'a list  *)

fun reverse xs
    = let fun walk nil
                = id
            | walk (x :: xs)
                = (walk xs) o (cons x)
      in walk xs nil
      end
```

The function space `'a list -> 'a list` arises because of three functional values: `id`, in one conditional branch; and, in the other, the results of consing an element and of calling `walk`.

We thus defunctionalize the program using a data type with three constructors and its associated apply function.

```
datatype 'a lam = LAM0
                | LAM1 of 'a
                | LAM2 of 'a lam * 'a lam

(*  apply : 'a lam * 'a list -> 'a list  *)
fun apply (LAM0, ys)
    = ys
  | apply (LAM1 x, ys)
    = x :: ys
  | apply (LAM2 (f, g), ys)
    = apply (f, apply (g, ys))
```

This data type makes it plain that in Hughes's monoid of intermediate lists, concatenation is performed in constant time (here with `LAM2`).

The rest of the defunctionalized program reads as follows.

```
(*  id_def : 'a lam  *)
val id_def = LAM0

(*  cons_def : 'a -> 'a lam  *)
fun cons_def x = LAM1 x

(*  o_def : 'a lam * 'a lam -> 'a lam  *)
fun o_def (f, g) = LAM2 (f, g)

(*  reverse_def : 'a list -> 'a list  *)
(*         walk : 'a list -> 'a lam   *)
fun reverse_def xs
    = let fun walk nil
                = id_def
            | walk (x :: xs)
                = o_def (walk xs, cons_def x)
      in apply (walk xs, nil)
      end
```

The auxiliary functions are only aliases for the data-type constructors. We also observe that `LAM1` and `LAM2` are always used in connection with each other. Therefore, they can be fused in a single constructor `LAM3` and so can their treatment in `apply_lam`. The result reads as follows.

```
datatype 'a lam_alt = LAM0
                    | LAM3 of 'a lam_alt * 'a

(*  apply_lam_alt : 'a lam_alt * 'a list -> 'a list  *)
fun apply_lam_alt (LAM0, ys)
    = ys
  | apply_lam_alt (LAM3 (f, x), ys)
    = apply_lam_alt (f, x :: ys)

(*  reverse_def_alt : 'a list -> 'a list     *)
(*             walk : 'a list -> 'a lam_alt  *)
fun reverse_def_alt xs
    = let fun walk nil
              = LAM0
            | walk (x :: xs)
              = LAM3 (walk xs, x)
      in apply_lam_alt (walk xs, nil)
      end
```

As in Section 2.1, we can see that `reverse_def_alt` embeds its input list into the data type `lam_alt`, homomorphically. The associated apply function could therefore be made to work directly on the input list. We also recognize `apply_lam_alt` as an uncurried version of the fast reverse function with an accumulator.

Hughes also uses his representation of intermediate lists to define a 'fields' function that extracts words from strings. His representation gives rise to an efficient implementation of the fields function. And indeed, as for reverse above, defunctionalizing this implementation gives the fast implementation that accumulates words in reverse order and reverses them using a fast reverse function once the whole word has been found. Defunctionalization thus confirms the effectiveness of Hughes's representation.

2.3 Defunctionalizing Church-encoded non-recursive data structures

Church-encoding a value amounts to representing it by a λ-term in such a way that operations on this value are carried out by applying the representation to specific λ-terms [3, 9, 24, 33].

A data structure is a sum in a domain. (When the data structure is inductive, the domain is recursive.) A sum is defined by its corresponding injection functions and a case dispatch [55, page 133]. Church-encoding a data structure consists in (1) combining injection functions and case dispatch into λ-terms and (2) operating by function application.

In the rest of this section, for simplicity, we only consider Church-encoded data structures that are uncurried. This way, we can defunctionalize them as a whole.

For example, monotyped Church pairs and their selectors are defined as follows.

```
(*  Church_pair : 'a * 'a -> ('a * 'a -> 'a) -> 'a  *)
fun Church_pair (x1, x2)
    = fn s : 'a * 'a -> 'a => s (x1, x2)

(*  Church_fst : (('a * 'a -> 'a) -> 'b) -> 'b  *)
fun Church_fst p
    = p (fn (x1, x2) => x1)

(*  Church_snd : (('a * 'a -> 'a) -> 'b) -> 'b  *)
fun Church_snd p
    = p (fn (x1, x2) => x2)
```

A pair is represented as a λ-term expecting one argument. This argument is a selector corresponding to the first or the second projection.

In general, each of the injection functions defining a data structure has the following form.

$$\mathrm{inj}_i = \lambda(x_1, ..., x_n).\underline{\lambda(s_1, ..., s_m).s_i\ (x_1, ..., x_n)}$$

So what happens if one defunctionalizes a Church-encoded data structure, i.e., the result of the injection functions? Each injection function gives rise to a data-type constructor whose arguments correspond to the free variables in the term underlined just above. These free variables are precisely the parameters of the injection functions, which are themselves the parameters of the original constructors that were Church encoded.

Therefore defunctionalizing Church-encoded data structures (i.e., the result of their injection functions) gives rise to the same data structures, prior to Church encoding. These data structures are accessed through the auxiliary apply functions introduced by defunctionalization.

For example, monotyped Church pairs and their selectors are defunctionalized as follows.

- The selectors are closed terms and therefore the corresponding constructors are parameterless. By definition, a selector is passed a tuple of arguments and returns one of them.

```
datatype sel = FST
             | SND

(*  apply_sel : sel * ('a * 'a) -> 'a  *)
fun apply_sel (FST, (x1, x2))
    = x1
  | apply_sel (SND, (x1, x2))
    = x2
```

- There is one injection function for pairs, and therefore it gives rise to a data type with one constructor for the values of the two free variables of the result of `Church_pair`. The corresponding apply function performs a selection. (N.B: `apply_pair` calls `apply_sel`, reflecting the curried type of `Church_pair`.)

```
datatype 'a pair = PAIR of 'a * 'a

(*  apply_pair : 'a pair * sel -> 'a  *)
fun apply_pair (PAIR (x1, x2), s)
    = apply_sel (s, (x1, x2))
```

- Finally, constructing a pair amounts to constructing a pair, à la Tarski one could say [21], and selecting a component of a pair is achieved by calling `apply_pair`, which in turns calls `apply_sel`.

```
(*  Church_pair_def : 'a * 'a -> 'a pair  *)
fun Church_pair_def (x1, x2)
    = PAIR (x1, x2)

(*  Church_fst_def : 'a pair -> 'a  *)
fun Church_fst_def p
    = apply_pair (p, FST)

(*  Church_snd_def : 'a pair -> 'a  *)
fun Church_snd_def p
    = apply_pair (p, SND)
```

An optimizing compiler would inline both apply functions. The resulting selectors, together with the defunctionalized pair constructor, would then coincide with the original definition of pairs, prior to Church encoding.

2.4 Defunctionalizing Church-encoded recursive data structures

Let us briefly consider Church-encoded binary trees. Two injection functions occur: one for the leaves, and one for the nodes. A Church-encoded tree is a λ-term expecting two arguments. These arguments are the selectors corresponding to whether the tree is a leaf or whether it is a node.

```
fun Church_leaf x
    = fn (s1, s2) => s1 x

fun Church_node (t1, t2)
    = fn (s1, s2) => s2 (t1 (s1, s2), t2 (s1, s2))
```

Due to the recursive nature of binary trees, `Church-node` propagates the selectors to the subtrees.

In general, each of the injection functions defining a data structure has the following form.

$$\text{inj}_i = \lambda(x_1, ..., x_n).\ \underline{\lambda(s_1, ..., s_m).s_i\ (x_1, ..., x_j\ (s_1, ..., s_m), ..., x_n)}$$

where $x_j\ (s_1, ..., s_m)$ occurs for each x_j that is in the data type.

So what happens if one defunctionalizes a Church-encoded recursive data structure, i.e., the result of the injection functions? Again, each injection function gives rise to a data-type constructor whose arguments correspond to the free variables in the term underlined just above. These free variables are precisely the parameters of the injection functions, which are themselves the parameters of the original constructors that were Church encoded.

Therefore defunctionalizing Church-encoded recursive data structures (i.e., the result of their injection functions) also gives rise to the same data structures, prior to Church encoding. These data structures are accessed through the auxiliary apply functions introduced by defunctionalization.

Let us get back to Church-encoded binary trees. Since defunctionalization is a whole-program transformation, we consider a whole program. Let us consider the function computing the depth of a Church-encoded binary tree. This function passes two selectors to its argument. The first is the constant function returning 0, and accounting for the depth of a leaf. The second is a function that will be applied to the depth of the subtrees of each node, and computes the depth of the node by taking the max of the depths of the two subtrees and adding one.

```
fun Church_depth t
    = t (fn x => 0,
         fn (d1, d2) => 1 + Int.max (d1, d2))
```

This whole program is defunctionalized as follows.

- The selectors give rise to two constructors, `SEL_LEAF` and `SEL_NODE`, and the corresponding two apply functions, `apply_sel_leaf` and `apply_sel_node`.

```
datatype sel_leaf = SEL_LEAF

fun apply_sel_leaf (SEL_LEAF, x)
    = 0

datatype sel_node = SEL_NODE

fun apply_sel_node (SEL_NODE, (d1, d2))
    = Int.max (d1, d2) + 1
```

- As for the injection functions, as noted above, they give rise to two constructors, `LEAF` and `NODE`, and the corresponding apply function.

```
datatype 'a tree = LEAF of 'a
                 | NODE of 'a tree * 'a tree

fun Church_leaf_def x
    = LEAF x

fun Church_node_def (t1, t2)
    = NODE (t1, t2)
```

- Finally, the defunctionalized main function applies its argument to the two selectors.

```
(*   depth_def : 'a tree -> int                          *)
(*  apply_tree : 'a tree * (sel_leaf * sel_node) -> int  *)
fun depth_def t
    = apply_tree (t, (SEL_LEAF, SEL_NODE))
and apply_tree (LEAF x, (sel_leaf, sel_node))
    = apply_sel_leaf (sel_leaf, x)
  | apply_tree (NODE (t1, t2), (sel_leaf, sel_node))
    = apply_sel_node (sel_node,
                      (apply_tree (t1,
                                   (sel_leaf, sel_node)),
                       apply_tree (t2,
                                   (sel_leaf, sel_node))))
```

Again, an optimizing compiler would inline both apply functions and `SEL_LEAF` and `SEL_NODE` would then disappear. The result would thus coincide with the original definition of binary trees, prior to Church encoding.

2.5 Church-encoding the result of defunctionalization

As can be easily verified with the Church pairs and the Church trees above, Church-encoding the result of defunctionalizing a Church-encoded data structure gives back this Church-encoded data structure: the apply functions revert to simple applications, the main data-structure constructors become injection functions, and the auxiliary data-structure constructors become selectors.

In practice, however, one often inlines selectors during Church encoding if they only occur once—which Shivers refers to as "super-beta" [45]. Doing so yields an actual inverse to defunctionalization, as illustrated in Section 4.3. This "refunctionalization" is used, e.g., in Danvy, Grobauer, and Rhiger's work on goal-directed evaluation [13]. We also illustrate it in Section 4.3.

In Church-encoded data structures, selectors have the flavor of a continuation. In the next section, we consider how to defunctionalize continuations.

3. Defunctionalization of CPS-Transformed First-Order Programs

As functional representations of control, continuations provide a natural target for defunctionalization. In this section, we investigate the process of transforming direct-style programs into continuation-passing style (CPS) programs [12, 48] and defunctionalizing their continuation. We then compare this process with Wand's continuation-based program-transformation strategies [53].

3.1 String parsing

We consider a recognizer for the language $0^n 1^n$. We write it as a function of type `int list -> bool`. The input is a list of integers, and the recognizer checks whether it is the concatenation of a list of n 0s and of a list of n 1s (and therefore of length $2n$).

We start with a recursive-descent parser traversing the input list.

```
(*  rec0 : int list -> bool      *)
(*  walk : int list -> int list  *)
fun rec0 xs
    = let exception NOT
          fun walk (0 :: xs')
              = (case walk xs'
                   of 1 :: xs''
                      => xs''
                    | _
                      => raise NOT)
            | walk xs
              = xs
      in (walk xs = nil) handle NOT => false
      end
```

The auxiliary function `walk` traverses the input list. Every time it encounters `0`, it calls itself recursively. When it meets something else than `0`, it returns the rest of the list, and expects to find `1` at every return. In case of mismatch (i.e., a list element other than `1` for returns, or a list that is too short or too long), an exception is raised.

Let us write `walk` in continuation-passing style [12, 48].

```
(*  rec1 : int list -> bool                          *)
(*  walk : int list * (int list -> bool) -> bool  *)
fun rec1 xs
    = let fun walk (0 :: xs', k)
              = walk (xs', fn (1 :: xs'')
                              => k xs''
                            | _
                              => false)
            | walk (xs, k)
              = k xs
      in walk (xs, fn xs' => xs' = nil)
      end
```

The auxiliary function `walk` traverses the input list tail-recursively (and thus does not need any exception). If it meets something else than `0`, it sends the current list to the current continuation. If it encounters `0`, it iterates down the list with a new continuation. If the new continuation is sent a list starting with `1`, it sends the tail of that list to the current continuation; otherwise, it returns `false`. The initial continuation tests whether it is sent the empty list.

Let us defunctionalize `rec1`. Two function abstractions occur: one for the initial continuation and one for intermediate continuations.

```
datatype cont = CONT0
              | CONT1 of cont

(*  apply2 : (cont * int list) -> bool  *)
fun apply2 (CONT0, xs')
    = xs' = nil
  | apply2 (CONT1 k, 1 :: xs'')
    = apply2 (k, xs'')
  | apply2 (CONT1 k, _)
    = false

(*  rec2 : int list -> bool         *)
(*  walk : int list * cont -> bool  *)
fun rec2 xs
    = let fun walk (0 :: xs', k)
              = walk (xs', CONT1 k)
            | walk (xs, k)
              = apply2 (k, xs)
      in walk (xs, CONT0)
      end
```

We identify the result as implementing a push-down automaton [26]. This automaton has two states and one element in the stack alphabet. The two states are represented by the two functions `walk` and `apply2`. The stack is implemented by the data type `cont`. The transitions are the tail-recursive calls. This automaton accepts an input if processing this input ends with an empty stack.

We also observe that `cont` implements Peano numbers. Let us replace them with ML integers.

```
(*  apply3 : (int * int list) -> bool  *)
fun apply3 (0, xs')
    = xs' = nil
  | apply3 (k, 1 :: xs'')
    = apply3 (k-1, xs'')
  | apply3 (k, _)
    = false

(*  rec3 : int list -> bool        *)
(*  walk : int list * int -> bool  *)
fun rec3 xs
    = let fun walk (0 :: xs', k)
              = walk (xs', k+1)
            | walk (xs, k)
              = apply3 (k, xs)
      in walk (xs, 0)
      end
```

The result is the usual iterative two-state recognizer with a counter.

In summary, we started from a first-order recursive version and we CPS-transformed it, making it higher-order and thus defunctionalizable. We identified that the defunctionalized program implements a push-down automaton. Noticing that the defunctionalized continuation implements Peano arithmetic, we changed its representation to built-in integers and we identified that the result is the usual iterative two-state recognizer with a counter.

3.2 Continuation-based program transformation strategies, reconsidered

Wand's classical work on continuation-based program transformation [53] suggests one (1) to CPS-transform a program; (2) to design a data-structure representation for the continuation; and (3) to use this representation to improve the initial program. We observe that in each of the examples mentioned in Wand's article, defunctionalization answers the challenge of finding a data structure representing the continuation—which is significant because finding such "data-structure continuations" was one of the motivations of the work. Nevertheless, defunctionalization is not considered in the textbooks and articles that refer to Wand's article.[1]

At any rate, Wand's work was seminal in that it showed how detouring via CPS yields iterative programs with accumulators. In addition, Reynolds's work shows how defunctionalizing the continuation of CPS-transformed programs gives rise to traditional, first-order accumulators.

[1]The textbooks and articles we are aware of include those found in the Research Index at `http://citeseer.nj.nec.com/`.

We also observe that defunctionalized continuations account for the call/return patterns of recursively defined functions. Therefore, as pointed out by Dijkstra in the late 1950's [16], they evolve in a stack-like fashion. A corollary of this remark is that before defunctionalization, continuations are also used LIFO when they result from the CPS transformation of a program that does not use control operators [10, 11, 12, 14, 15, 39, 40, 48].

4. A Syntactic Theory in the Light of Defunctionalization

In this section, we present a definitional interpreter for a syntactic theory [19, 56] of arithmetic expressions. We observe that this definitional interpreter corresponds to the output of defunctionalization. We present the corresponding higher-order interpreter, which is in continuation-passing style. Its continuation represents the evaluation context of the syntactic theory.

4.1 A syntactic theory

We consider a simplified language of arithmetic expressions. An arithmetic expression is either a value (a literal) or a computation. A computation is either an addition or a conditional expression testing whether its first argument is zero.

$$e \quad ::= \quad n \mid e + e \mid \textsc{ifz}\ e\ e\ e$$

A syntactic theory provides a reduction relation on expressions by defining values, evaluation contexts, and redexes [19].

The values are literals, and the evaluation contexts are defined as follows.

$$\begin{array}{lcl} v & ::= & n \\ E & ::= & [\,] \mid E[[\,] + e] \mid E[v + [\,]] \mid E[\textsc{ifz}\ [\,]\ e\ e] \end{array}$$

Plugging an expression e into a context E, written $E[e]$, is defined as follows.

$$\begin{array}{rcl} ([\,])[e] & = & e \\ (E[[\,] + e'])[e] & = & E[e + e'] \\ (E[v + [\,]])[e] & = & E[v + e] \\ (E[\textsc{ifz}\ [\,]\ e_1\ e_2])[e] & = & E[\textsc{ifz}\ e\ e_1\ e_2] \end{array}$$

The reduction relation is then defined by the following rules, where the expressions plugged into the context on the left-hand side are called redexes.

$$\begin{array}{rcll} E[n_1 + n_2] & \rightarrow & E[n_3] & \text{where } n_3 \text{ is the sum of } n_1 \text{ and } n_2 \\ E[\textsc{ifz}\ 0\ e_1\ e_2] & \rightarrow & E[e_1] & \\ E[\textsc{ifz}\ n\ e_1\ e_2] & \rightarrow & E[e_2] & \text{if } n \neq 0 \end{array}$$

These definitions satisfy a "unique decomposition" lemma [19, 56]: any expression, e, that is not a value can be uniquely decomposed into an evaluation context, E, and a redex, r, such that $e = E[r]$.

4.2 Implementation

Arithmetic expressions are defined with the following data type.

```
datatype ae = V of int
            | C of comp
   and comp = ADD of ae * ae
            | IFZ of ae * ae * ae
```

In `ae`, we distinguish between values (literals) and computations (additions and conditional expressions), as traditional.

Evaluation contexts are defined with the following data type.

```
datatype ec = EMPTY
            | ADD1 of ec * ae
            | ADD2 of ec * int
            | IFZ0 of ec * ae * ae
```

The corresponding plugging function reads as follows.

```
(*  plug : ec * ae -> ae  *)
fun plug (EMPTY, e)
    = e
  | plug (ADD1 (x, e2), e)
    = plug (x, C (ADD (e, e2)))
  | plug (ADD2 (x, i1), e)
    = plug (x, C (ADD (V i1, e)))
  | plug (IFZ0 (x, e1, e2), e)
    = plug (x, C (IFZ (e, e1, e2)))
```

A computation undergoes a reduction step when (1) it is decomposed into a redex and its context, (2) the redex is contracted, and (3) the result is plugged into the context.

```
(*  reduce1 : comp * ec -> ae  *)
fun reduce1 (ADD (V i1, V i2), x)
    = plug (x, V (i1+i2))
  | reduce1 (ADD (V i1, C c2), x)
    = reduce1 (c2, ADD2 (x, i1))
  | reduce1 (ADD (C c1, e2), x)
    = reduce1 (c1, ADD1 (x, e2))
  | reduce1 (IFZ (V 0, e1, e2), x)
    = plug (x, e1)
  | reduce1 (IFZ (V i, e1, e2), x)
    = plug (x, e2)
  | reduce1 (IFZ (C c0, e1, e2), x)
    = reduce1 (c0, IFZ0 (x, e1, e2))
```

Evaluation is specified by repeatedly performing a reduction until a value is obtained.

```
(*  eval : ae -> int  *)
fun eval (V i)
    = i
  | eval (C c)
    = eval (reduce1 (c, EMPTY))
```

4.3 Refunctionalization

We observe that the program above precisely corresponds to the output of defunctionalization: `plug` is the apply function of `ec`. The corresponding input to defunctionalization thus reads as follows.

```
(*  reduce1 : comp * (ae -> 'a) -> 'a  *)
fun reduce1 (ADD (V i1, V i2), x)
    = x (V (i1+i2))
  | reduce1 (ADD (V i1, C c2), x)
    = reduce1 (c2, fn e2 => x (C (ADD (V i1, e2))))
  | reduce1 (ADD (C c1, e2), x)
    = reduce1 (c1, fn e1 => x (C (ADD (e1, e2))))
  | reduce1 (IFZ (V 0, e1, e2), x)
    = x e1
  | reduce1 (IFZ (V i, e1, e2), x)
    = x e2
  | reduce1 (IFZ (C c0, e1, e2), x)
    = reduce1 (c0, fn e0 => x (C (IFZ (e0, e1, e2))))

(*  eval : ae -> int  *)
fun eval (V i)
    = i
  | eval (C c)
    = eval (reduce1 (c, fn e => e))
```

We observe that `reduce1` is written in continuation-passing style. Its continuation therefore represents the evaluation context of the syntactic theory.

4.4 Back to direct style

Since `reduce1` uses its continuation canonically, it can be mapped back to direct style [10, 14]. The direct-style version of `reduce1` reads as follows.

```
(*  reduce1 : comp -> ae  *)
fun reduce1 (ADD (V i1, V i2))
    = V (i1+i2)
  | reduce1 (ADD (V i1, C c2))
    = C (ADD (V i1, reduce1 c2))
  | reduce1 (ADD (C c1, e2))
    = C (ADD (reduce1 c1, e2))
  | reduce1 (IFZ (V 0, e1, e2))
    = e1
  | reduce1 (IFZ (V i, e1, e2))
    = e2
  | reduce1 (IFZ (C c0, e1, e2))
    = C (IFZ (reduce1 c0, e1, e2))

(*  eval : ae -> int  *)
fun eval (V i)
    = i
  | eval (C c)
    = eval (reduce1 c)
```

The result is a definitional interpreter with an implicit representation of contexts.

4.5 Summary and conclusion

We have considered a naive definitional interpreter for a syntactic theory, and have observed that the contexts and their plugging function are the defunctionalized counterpart of a continuation. This observation has led us to implement the definitional interpreter in direct style. (In that sense, Sections 3 and 4 are symmetric, since Section 3 starts with a direct-style program and ends with a defunctionalized CPS program.)

One may wonder how the various representations of contexts in a syntactic theory influence reasoning about programs. In the next section, we compare two correctness proofs of a program, before and after defunctionalization.

5. A Comparison between Correctness Proofs before and after Defunctionalization: Matching Regular Expressions

We consider a traditional continuation-based matcher for regular expressions [26], we defunctionalize it, and we compare and contrast its correctness proof before and after defunctionalization. To this end, Section 5.1 briefly reviews regular expressions and the languages they represent; Section 5.2 presents the continuation-based matcher, which is higher-order, and its defunctionalized counterpart; and Section 5.3 compares and contrasts their correctness proofs.

5.1 Regular expressions

The grammar for regular expressions, r, over the alphabet Σ and the corresponding language, $\mathcal{L}(r)$, are as follows.

$$
\begin{array}{rcrcl}
r & ::= & \mathbf{0} & \mathcal{L}(\mathbf{0}) & = & \emptyset \\
 & | & \mathbf{1} & \mathcal{L}(\mathbf{1}) & = & \{\epsilon\} \\
 & | & \mathbf{c} & \mathcal{L}(\mathbf{c}) & = & \{\mathbf{c}\} \quad \text{where } \mathbf{c} \in \Sigma \\
 & | & r\ r & \mathcal{L}(r_1\ r_2) & = & \mathcal{L}(r_1)\mathcal{L}(r_2) \\
 & | & r{+}r & \mathcal{L}(r_1{+}r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
 & | & r^* & \mathcal{L}(r^*) & = & \mathcal{L}(r)^* = \bigcup_{i \in \omega} (\mathcal{L}(r))^i
\end{array}
$$

We represent strings as lists of ML characters, and regular expressions as elements of the following ML datatype.

```
datatype regexp = ZERO | ONE | CHAR of char
                | CAT of regexp * regexp
                | SUM of regexp * regexp
                | STAR of regexp
```

We define the corresponding notion of "the language of a regular expression" as follows.

$$
\begin{array}{rcl}
\mathcal{L}(\texttt{ZERO}) & = & \{\} \\
\mathcal{L}(\texttt{ONE}) & = & \{\texttt{nil}\} \\
\mathcal{L}(\texttt{CHAR c}) & = & \{[\texttt{c}]\} \\
\mathcal{L}(\texttt{CAT(r1, r2)}) & = & \mathcal{L}(\texttt{r1})\mathcal{L}(\texttt{r2}) \\
\mathcal{L}(\texttt{SUM(r1, r2)}) & = & \mathcal{L}(\texttt{r1}) \cup \mathcal{L}(\texttt{r2}) \\
\mathcal{L}(\texttt{STAR r}) & = & \bigcup_{i \in \omega} (\mathcal{L}(\texttt{r}))^i
\end{array}
$$

The concatenation of languages is defined as $L_1 L_2 = \{\texttt{x@y} \mid \texttt{x} \in L_1 \wedge \texttt{y} \in L_2\}$, where we use the append function (noted `@` as in ML) to concatenate strings.

5.2 The two matchers

Our reference matcher for regular expressions is higher-order (Figure 1). We then present its defunctionalized counterpart (Figure 2).

5.2.1 The higher-order matcher

Figure 1 displays our reference matcher, which is compositional and continuation-based. Compositional: all recursive calls to `accept` operate on a proper subpart of the regular-expression under consideration. And continuation-based: the control flow of the matcher is driven by continuations.

The main function is `match`. It is given a regular expression and a list of characters, and calls `accept` with the regular expression, the list, and an initial continuation expecting a list of characters and testing whether this list is empty.

The `accept` function recursively descends its input regular expression, threading the list of characters.

The `accept_star` function is a lambda-lifted version of a recursive continuation defined locally in the `STAR` branch. (The situation is exactly the same as in a compositional matcher for an imperative language with while loops. There, one writes an auxiliary recursive function as well.) This recursive continuation checks that matching has progressed in the string.

Recently, Harper has published a similar matcher to illustrate "proof-directed debugging" [25]. Playfully, he considered a non-compositional matcher that does not check progress when matching a Kleene star. His article shows (1) how one stumbles on the non-compositional part when attempting a proof by structural induction; and (2) how one realizes that the matcher diverges for pathological regular expressions such as `STAR ONE`. Harper then (1) makes his matcher compositional and (2) normalizes regular expressions to exclude pathological regular expressions. Instead, we start from a compositional matcher and we include a progress test in `accept_star`, which lets us handle pathological regular expressions.

```
(*  accept      : regexp * char list * (char list -> bool) -> bool  *)
(*  accept_star : regexp * char list * (char list -> bool) -> bool  *)
fun accept (r, s, k)
    = (case r of ZERO         => false
               | ONE          => k s
               | CHAR c       => (case s of (c'::s')  => c = c' andalso k s'
                                          | nil       => false)
               | CAT (r1, r2) => accept (r1, s, fn s' => accept (r2, s', k))
               | SUM (r1, r2) => accept (r1, s, k) orelse accept (r2, s, k)
               | STAR r'      => accept_star (r', s, k))
and accept_star (r, s, k)
    = k s orelse accept (r, s, fn s' => not (s = s') andalso accept_star (r, s', k))

(*  match : regexp * char list -> bool  *)
fun match (r, s)
    = accept (r, s, fn s' => s' = nil)
```

Figure 1: Higher-order, continuation-based matcher for regular expressions

```
datatype regexp_stack = EMPTY | ACCEPT of regexp * regexp_stack | ACCEPT_STAR of char list * regexp * regexp_stack

(*  accept_def      : regexp * char list * regexp_stack -> bool  *)
(*  accept_star_def : regexp * char list * regexp_stack -> bool  *)
(*  pop_and_accept  : regexp_stack * char list                   *)
fun accept_def (r, s, k)
    = (case r of ZERO         => false
               | ONE          => pop_and_accept (k, s)
               | CHAR c       => (case s of (c'::s') => c = c' andalso pop_and_accept (k, s')
                                          | nil      => false)
               | CAT (r1, r2) => accept_def (r1, s, ACCEPT (r2, k))
               | SUM (r1, r2) => accept_def (r1, s, k) orelse accept_def (r2, s, k)
               | STAR r'      => accept_star_def (r', s, k))
and accept_star_def (r, s, k)
    = pop_and_accept (k, s) orelse accept_def (r, s, ACCEPT_STAR (s, r, k))
and pop_and_accept (EMPTY, s')
    = s' = nil
  | pop_and_accept (ACCEPT (r2, k), s')
    = accept_def (r2, s', k)
  | pop_and_accept (ACCEPT_STAR (s, r, k), s')
    = not (s = s') andalso accept_star_def (r, s', k)

(*  match : regexp * char list -> bool  *)
fun match (r, s)
    = accept_def (r, s, EMPTY)
```

Figure 2: First-order, stack-based matcher for regular expressions

5.2.2 The first-order matcher

Defunctionalizing the matcher of Figure 1 yields a data type representing the continuations and its associated apply function.

The data type represents a stack of regular expressions (possibly with a side condition for the test in Kleene stars). The apply function merely pops the top element off this stack and tries to match it against the rest of the string. We thus name the data type "`regexp_stack`" and the apply function "`pop_and_accept`". We also give a meaningful name to the datatype constructors. Figure 2 displays the result.

5.3 The two correctness proofs

We give a correctness proof of both the higher-order version and the first-order version, and we investigate whether each proof can be converted to a proof for the other version.

The correctness criterion we choose is simply that for all regular expressions `r` and strings `s` (represented by a list of characters),

$$\begin{cases} \texttt{match (r, s)} \text{ terminates, and} \\ \texttt{s} \in \mathcal{L}(\texttt{r}) \;\Leftrightarrow\; \texttt{match (r, s)} \rightsquigarrow \texttt{true} \end{cases}$$

When writing `match (r, s)` $\rightsquigarrow$ `true`, we mean that evaluating `match (r, s)` terminates and yields the result `true`. More generally we equate expressions if they evaluate to the same result (and therefore divergent expressions are not equivalent to anything). We also reason about ML programs equationally.

5.3.1 Correctness proof of the higher-order matcher

Since `match (r, s)` = `accept (r, s, fn s' => s' = nil)`, by definition, it is sufficient to prove that for `s` and `r` as above, and for any function from lists of characters to booleans terminating on all suffixes of `s`, denoted by `k`,

$$\begin{cases} \texttt{accept (r, s, k)} \text{ terminates, and} \\ \texttt{s} \in \mathcal{L}(\texttt{r})\mathcal{L}(\texttt{k}) \;\Leftrightarrow\; \texttt{accept (r, s, k)} \rightsquigarrow \texttt{true} \end{cases}$$

where we define the language of a "string-acceptor" `k` as the set $\{\texttt{s} \mid \texttt{k s} \rightsquigarrow \texttt{true}\}$.

The proof is by structural induction on the regular expression. In the case where `r = STAR r'`, a subproof shows that the following holds for any string.

$$\begin{cases} \texttt{accept\_star (r', s, k)} \text{ terminates, and} \\ \texttt{s} \in \mathcal{L}(\texttt{r'})^*\mathcal{L}(\texttt{k}) \;\Leftrightarrow\; \texttt{accept\_star (r', s, k)} \rightsquigarrow \texttt{true} \end{cases}$$

The subproof is by well-founded induction on the structure of the string (suffixes are smaller) for the "$\Leftarrow$" direction, and by mathematical induction on the natural number n such that $\texttt{s} \in \mathcal{L}(\texttt{r'})^n\mathcal{L}(\texttt{k})$ for the "$\Rightarrow$" direction. Both subproofs use the outer induction hypothesis for `accept (r', s, k)`.

We can transfer this proof to the defunctionalized version. Since `k s` translates to `pop_and_accept (k, s)`, we define $\mathcal{L}(\texttt{k})$ to read $\{\texttt{s} \mid \texttt{pop\_and\_accept (k, s)} \rightsquigarrow \texttt{true}\}$. The proof then goes through in exactly the same format.

5.3.2 Correctness proof of the first-order matcher

Alternatively, if we were to prove the correctness of the first-order matcher directly, we would be less inclined to recognize the stack `k` as representing a function. Instead, we could easily end up proving the following three propositions by mutual induction.

$$P_1(\texttt{r},\texttt{s},\texttt{k}) := \texttt{accept (r, s, k)} \rightsquigarrow \texttt{true} \;\Leftrightarrow\; \texttt{s} \in \mathcal{L}(\texttt{r})\mathcal{L}(\texttt{k})$$

$$P_2(\texttt{k},\texttt{s}) := \texttt{pop\_and\_accept (k, s)} \rightsquigarrow \texttt{true} \;\Leftrightarrow\; \texttt{s} \in \mathcal{L}(\texttt{k})$$

$$P_3(\texttt{r},\texttt{s},\texttt{k}) := \texttt{accept\_star (r, s, k)} \rightsquigarrow \texttt{true} \;\Leftrightarrow\; \texttt{s} \in \mathcal{L}(\texttt{r})^*\mathcal{L}(\texttt{k})$$

where we define the language of a stack of regular expressions as follows.

$$\begin{aligned} \mathcal{L}(\texttt{EMPTY}) &= \{\texttt{nil}\} \\ \mathcal{L}(\texttt{ACCEPT (r, k)}) &= \mathcal{L}(\texttt{r})\mathcal{L}(\texttt{k}) \\ \mathcal{L}(\texttt{ACCEPT\_STAR (s, r, k)}) &= (\mathcal{L}(\texttt{r})^*\mathcal{L}(\texttt{k})) \setminus \{\texttt{s}\} \end{aligned}$$

For brevity we ignore the termination part of the proof and assume that all the functions are total. We prove, by well-founded induction on the propositions themselves, that P_1, P_2, and P_3 hold for any choices of `s`, `r`, and `k`. The ordering is an intricate mapping into $\omega \times \omega$, ordered lexicographically so that the proof of a proposition only depends on "smaller" propositions.

This proof is more convoluted than the higher-order one for two reasons:

1. it separates the language of the continuation from the function that matches it, so one has to check whether the function really matches the correct language; and
2. it combines the two nested inductions of the higher-order proof into one well-founded induction.

Still this proof reveals a property of the continuations in the higher-order version, namely that there are at most three different kinds of continuations in use, something that cannot be seen from the type of the continuation—a full function space.

We could thus define a subset of this function space inductively, so that it only contains the functions that can be generated by the three abstractions. The first-order proof could then be extended to the higher-order program by assuming everywhere that the continuation `k` lies in this subset and showing that newly generated continuations do too. In effect, the set of continuations is partitioned into disjoint subsets, just as the first-order datatype represents a sum, and then we can prove something about elements in each part.

5.4 Summary and conclusion

We have considered a matcher for regular expressions, both in higher-order form and in first-order form, and we have compared them and their correctness proof. The difference between the function-based and the datatype-based representation of continuations is reminiscent of the concept of 'junk' in algebraic semantics [22]. One representation is a full function space where many elements do not correspond to an actual continuation, and the other representation only contains elements corresponding to actual continuations. This difference finds an echo in the correctness proofs of the two matchers. In the first we cannot know what the function will do, and in the second we can inspect the value. On the other hand, we could make assumptions about the functions by restricting the function space (even if only in the proof) to a smaller one that is still closed under the constructions used.

Another difference between the two proofs is where the language of a continuation is treated. In the first-order case, a continuation can be inspected, so we can prove something for all continuations independently of where each one is defined. In the higher-order case, however, the only place where we know anything about a continuation is where it is constructed, because in the proof, we can look inside the abstraction where this continuation appears textually.

More generally, this section also illustrates that defunctionalizing a functional interpreter for a backtracking language that uses success continuations yields a recursive interpreter with one stack [13, 31, 41]. Similarly, defunctionalizing a functional interpreter that uses success and failure continuations yields an iterative interpreter with two stacks [13, 34].

6. Conclusions and Issues

Reynolds's defunctionalization technique connects the world of higher-order programs and the world of first-order programs. In this article, we have illustrated this connection by considering a variety of situations where defunctionalization proves fruitful in a declarative setting.

Higher-order functions provide a convenient support for specifying and for transforming programs. As we have shown, defunctionalization can lead to more concrete specifications, e.g., that use first-order accumulators. And as we have seen with Wand's continuation-based program-transformation strategies, defunctionalization can automate eureka steps to represent data-structure continuations.

Conversely, defunctionalization also increases one's awareness that some first-order programs naturally correspond to other, higher-order, programs. For example, functional interpreters for backtracking languages are variously specified with one or two control stacks and with one or two continuations, but these specifications are not disconnected, since defunctionalizing the continuation-based interpreters yields the corresponding stack-based ones. On a related note, CPS-transforming an interpreter with one continua-

tion is already known to automatically yield an interpreter with two continuations [12]. We are, however, not aware of a similar transformation for their stack-based counterparts.

We also have compared and contrasted the correctness proofs of a program, before and after defunctionalization. We have found that while the first-order and the higher-order programming methods suggest different proof methods, each of the proofs can be adapted to fit the other version of the program.

Finally, we have pointed out at the type-theoretical foundations of defunctionalization.

Acknowledgments

Andrzej Filinski and David Toman provided most timely comments on an earlier version of this article. This work has also benefited from the anonymous reviewers's attention as well as from Daniel Damian, Julia Lawall, Karoline Malmkjær, and Morten Rhiger's comments.

We would also like to thank John Reynolds for his encouraging words and Harald Søndergaard, our program chairman, for his patience.

7. References

[1] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In M. J. O'Donnell and S. Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, Jan. 1989. ACM Press.

[2] A. Banerjee, N. Heintze, and J. G. Riecke. Semantics-based design and correctness of control-flow analysis-based program transformations. Unpublished, Mar. 2001.

[3] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, 1984. Revised edition.

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

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

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

[7] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In Smolka [46], pages 56–71.

[8] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[9] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[10] O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[11] O. Danvy. Formalizing implementation strategies for first-class continuations. In Smolka [46], pages 88–103.

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

[13] O. Danvy, B. Grobauer, and M. Rhiger. A unifying approach to goal-directed evaluation. In W. Taha, editor, *Proceedings of the Second Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001)*, Lecture Notes in Computer Science, Florence, Italy, Sept. 2001. Springer-Verlag. To appear.

[14] O. Danvy and J. L. Lawall. Back to direct style II: First-class continuations. In W. 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.

[15] O. Danvy and F. Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Feb. 1995.

[16] E. W. Dijkstra. Recursive programming. In S. Rosen, editor, *Programming Systems and Languages*, chapter 3C, pages 221–227. McGraw-Hill, New York, 1960.

[17] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, Apr. 1987. Technical Report #87-011.

[18] L. Fegaras. lambda-DB. Available online at `http://lambda.uta.edu/lambda-DB/manual/`, 1999-2001.

[19] M. Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, Aug. 1987.

[20] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[21] J.-Y. Girard. Locus solum. *Mathematical Structures in Computer Science*, 11(3), 2001. To appear.

[22] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume IV, pages 80–149. Prentice-Hall, 1978.

[23] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[24] M. Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.

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

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[27] J. Hughes. Super combinators: A new implementation method for applicative languages. In D. P. Friedman and D. S. Wise, editors, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, Pennsylvania, Aug. 1982. ACM Press.

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

[29] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, Sept. 1985. Springer-Verlag.

[30] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[31] C. Mellish and S. Hardy. Integrating Prolog in the POPLOG environment. In J. A. Campbell, editor, *Implementations of PROLOG*, pages 147–162. Ellis Horwood, 1984.

[32] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In G. L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg Beach, Florida, Jan. 1996. ACM Press.

[33] T. A. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992.

[34] T. Nicholson and N. Y. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11(4):650–665, Oct. 1989.

[35] L. R. Nielsen. A denotational investigation of defunctionalization. Progress report (superseded by [36]), BRICS PhD School, University of Aarhus, June 1999.

[36] L. R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Dec. 2000.

[37] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[38] S. L. Peyton Jones and D. R. Lester. *Implementing Functional Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1992.

[39] J. Polakow. Linear logic programming with ordered contexts. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of the Second International Conference on Principles and Practice of Declarative Programming*, pages 68–79, Montréal, Canada, Sept. 2000. ACM Press.

[40] J. Polakow and K. Yi. Proving syntactic properties of exceptions in an ordered logical framework. In H. Kuchen and K. Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 61–77, Tokyo, Japan, Mar. 2001. Springer-Verlag.

[41] T. A. Proebsting. Simple translation of goal-directed evaluation. In R. K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 1–6, Las Vegas, Nevada, June 1997. ACM Press.

[42] J. C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, The Netherlands, 1982. North-Holland.

[43] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

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

[45] O. 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] G. Smolka, editor. *Proceedings of the Ninth European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany, Mar. 2000. Springer-Verlag.

[47] P. A. Steckler and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, Jan. 1997.

[48] G. L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[49] C. Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000.

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

[51] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, 1979.

[52] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Comput. Sci.*, 73(2):231–248, 1989.

[53] M. Wand. Continuation-based program transformation strategies. *J. ACM*, 27(1):164–180, Jan. 1980.

[54] D. C. Wang and A. W. Appel. Type-safe garbage collectors. In H. R. Nielson, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 166–178, London, United Kingdom, Jan. 2001. ACM Press.

[55] G. Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

[56] Y. Xiao, A. Sabry, and Z. M. Ariola. From syntactic theories to interpreters: Automating proofs of decomposition lemma. *Higher-Order and Symbolic Computation*, 14(4), 2001. To appear.