An Analytical Approach to Programs as Data Objects

Olivier Danvy
BRICS*
Department of Computer Science
University of Aarhus†

April 14, 2006

Abstract

This essay accompanies a selection of 32 articles (referred to in bold face in the text and marked with "$\surd$" in the bibliographic references) submitted to the University of Aarhus towards a *Doctor Scientarum* degree in Computer Science. The author's previous academic degree, beyond a doctoral degree in June 1986, is an *Habilitation à diriger les recherches* from the Université Pierre et Marie Curie (Paris VI) in France; the corresponding material was submitted in September 1992 and the degree was obtained in January 1993. The present 32 articles have all been written since 1993 and while at DAIMI. Except for one other PhD student, all co-authors are or have been the author's students here in Aarhus.

*Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.
†IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark.
Email: `danvy@daimi.au.dk`
Home page: `http://www.daimi.au.dk/~danvy`

Technical summary

This work presents an analytical approach to programs as data objects, i.e., to programs whose representation is operated upon by other programs. The approach is analytical in that a variety of program manipulations is considered and their impact on program understanding, design, and efficiency is measured.

Most of the programs considered here are language processors, i.e., programs interpreting, translating, or transforming other programs. As for the program manipulations, they concern block structure and lexical scope, the representation of functions at run time, and the representation of control.

Teknisk résumé

Dette arbejde præsenterer en analytisk tilgang til programmer som dataobjekter, dvs, til programmer hvis repræsentation bliver behandlet af andre programmer. Tilgangen er analytisk i og med at et udvalg af programmanipulationer bliver betragtet og deres indflydelse på program-forståelse, -design og -effektivitet bliver vurderet.

De fleste programmer, der betragtes her, er sprogbehandlingsprogrammer, dvs, programmer der fortolker, oversætter eller transformerer andre programmer. Hvad angår programmanipulationerne, vedrører de blokstruktur og leksikalsk virkefelt, repræsentation af funktioner på køretidspunktet og repræsentation af kontrol.

Acknowledgments

Not everything that counts can be counted,
and not everything that can be counted counts.
Albert Einstein

Ever since I joined BRICS, I have both benefited and learned from its 'three musketeers': Glynn Winskel for his scientific vision and academic encouragement, Mogens Nielsen for his kaleidoscopic competences and overall leadership, Erik Meineche Schmidt for his professional experience and organizational foresight, and Uffe Engberg for his practical sense and kind efficiency.

My heartfelt thanks also go to the people who make DAIMI run in practice: its secretarial and accounting team extraordinaire, its outstanding librarians, and its chivalrous software and hardware staff members.

A distinctive trait of DAIMI is its studious atmosphere, and this atmosphere is nurtured by its faculty members. I am grateful to each of them for it all, starting with Flemming and Hanne Nielson who suggested I apply to DAIMI in the first place. It has been, in particular, a privilege to co-teach dSem and dSprogSem with Glynn Winskel, Andrzej Filinski, and Peter Mosses; a joy to co-teach graduate courses with Erik Ernst, Mayer Goldberg, Ulrik Pagh Schultz, and Mads Torgersen; a boost to work with the dOvs, dSem, dSprogSem, and dProgSprog teaching assistants (as well as a reward to see several of them pursue a career in research or even academia later on, to say nothing of our students); and, pedagogically, an eye opener to have Peter Kornerup and then Neil Jones and Torben Mogensen as external evaluators for dOvs. I have also enjoyed very much the Pearls of Programming course held in the spring of 2000: it was a wonderful way to get to know each of its lecturers.

Yet what would be DAIMI, as well as BRICS, without its students (native or imported, as well as graduate or undergraduate)? Thanks to them, I have not learned only from my teachers and my colleagues. This goes especially for the students who somehow chose to work with me more closely. In fairness, though, this learning phenomenon did start earlier, and not only in Denmark, but also in the US and in France. To all of these students, including the earlier ones, I am grateful. (And I miss them too; they are all over the world now.)

Still, nothing exists in a vacuum and for that reason these words of thanks extend to my mentors, colleagues, co-authors, co-editors, and friends outside DAIMI as well as to my family—both the French part, the Danish part, and its intersection.

Contents

0 Introduction

One person's program
is another program's data.

There is something self-evident, for a computer scientist who has implemented a language processor (e.g., an interpreter, a compiler, or a pretty-printer), to consider a program as a data object. In contrast, a logician, for example, is considerably more wary of the advanced notions of level and meta-level in a description [289, Chapter 2]. The reason is that in computer science, these notions are basic, hands-on ones: undergraduate students are thrown into the cauldron, so to speak, when they are required to implement a compiler. Because of this requirement, computer scientists become mindful of the notions of level and meta-level very early in their career. In fact, many of them experienced a jolt, if not an enlightenment, when they realized that The Bug was not in their compiler, but in their test program.

What is less evident to a computer scientist—and the topic of the present essay—is that manipulating programs as data objects can shed a constructive light on their design. Indeed the author has found time and again that many programming artifacts, i.e., man-made constructions, were developed independently but actually correspond to each other via an independently known program transformation. This approach is not merely a reductionist one about the same elephant:

- it can be viewed as a comforting confirmation of the value of each of the independent artifacts,
- it can be used to check whether the entirety of one artifact is actually reflected in another one or whether this other one can be improved, and
- it can be used to construct new artifacts.

The domain of discourse here is call-by-value functional programming with effects, and the object of discourse is the associated notion of computation. We analyze a variety of semantic artifacts characterizing this notion of computation (calculi, abstract machines, and normalization functions) and a variety of program transformations over the representation of these artifacts.

Prerequisites and notation: A basic familiarity with Standard ML and Scheme, and with the notions of continuation and of partial evaluation, is expected from the reader.

Overview: This document is organized in four parts. The first part (Section 1) reviews the domain of discourse and the tools used here: functional languages, continuation-passing style, and continuations. The second part (Section 2) addresses program transformation: lambda-lifting and lambda-dropping, defunctionalization and refunctionalization, and CPS and direct-style transformations. The third part (Section 3) considers program execution and how various independently developed semantic artifacts correspond to each other by program transformation. The last part (Sections 4, 5, and 6) mix program transformation and program execution, and addresses partial evaluation, type-directed partial evaluation, and normalization by evaluation.

1 Tools

This section successively reviews functional programming, continuation-passing style (CPS), first-class continuations, and the CPS hierarchy, using Standard ML as the language of discourse.

1.1 Functional programming

"The withering away of the statement"
is a phenomenon we may live to see.
Peter J. Landin, 1965 [246]

In a functional language, the syntactic unit is the *expression*, the programming abstraction is the *function*, and the notion of execution is *evaluation*. For example, the following elementary function decrements an integer:

```
fn n => n - 1  (* : int -> int *)
```

This function is a parameterized expression [318], and its formal parameter `n` is instantiated by applying this function to an argument. For example, applying this function to `10`, i.e., evaluating the expression "`(fn n => n - 1) 10`" yields `9`.

For convenience, functions are often named:

```
fun sub1 n  (* sub1 : int -> int *)
    = n - 1
```

So for example, evaluating "`sub1 10`" yields `9`.

Here are two other auxiliary functions: the first one computes the product of two given integers, and the other one tests whether a given integer is `0`.

```
fun mul (n1, n2)                                  (* mul : int * int -> int *)
    = n1 * n2

fun zerop n                                          (* zerop : int -> bool *)
    = n = 0
```

To visualize evaluation, it is often convenient to trace its successive steps, writing vertical bars at the beginning of a line to indicate the current number of nested calls. For example, evaluating

```
mul (sub1 3, mul (3, sub1 5))
```

gives rise to the following trace:

```
mul (sub1 3, mul (3, sub1 5))
|sub1 3
|2
mul (2, mul (3, sub1 5))
|mul (3, sub1 5)
||sub1 5
||4
|mul (3, 4)
|12
mul (2, 12)
24
```

1.1.1 Induction and recursion

Under call-by-value evaluation, data are constructed inductively and processed recursively [252, 368]. For example, natural numbers in the Peano style are constructed from a base case, 0, and through an inductive case, as the successor of a natural number:

$$n ::= 0 \mid n + 1$$

The following identities match this inductive construction:

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x \times x^n \end{aligned}$$

Based on these identities, and using the auxiliary functions above, one can define an exponentiation function `power0` that given an integer x and a natural number n, computes x^n by peeling off an increment until the base case is reached. As an aid to the eye, the initial call and the recursive call to `visit` are shaded:

```
(*  power0 : int * int -> int  *)
fun power0 (x, n)
    = let fun visit n                                    (* visit : int -> int *)
              = if zerop n
                then 1
                else mul (x, visit (sub1 n))
      in visit n
      end
```

The following trace visualizes the computation of 10^2. For brevity, the auxiliary functions are omitted and as an aid to the eye, the calls to `visit` are shaded as well as their result:

```
power0 (10, 2)
|visit 2
||if zerop 2 then 1 else mul (10, visit (sub1 2))
|||zerop 2
|||false
||if false then 1 else mul (10, visit (sub1 2))
||mul (10, visit (sub1 2))
|||visit (sub1 2)
||||sub1 2
||||1
|||visit 1
||||if zerop 1 then 1 else mul (10, visit (sub1 1))
|||||zerop 1
|||||false
||||if false then 1 else mul (10, visit (sub1 1))
||||mul (10, visit (sub1 1))
|||||visit (sub1 1)
||||||sub1 1
||||||0
|||||visit 0
||||||if zerop 0 then 1 else mul (10, visit (sub1 0))
|||||||zerop 0
|||||||true
||||||if true then 1 else mul (10, visit (sub1 0))
||||||1
|||||1
```

```
||||mul (10, 1)
||||10
|||10
||mul (10, 10)
||100
|100
100
```

The maximum number of nested calls (i.e., of bars at the beginning of a line in the trace) depends on the input.

1.1.2 Tail recursion

Tail calls, i.e., calls that occur last in the course of evaluating the body of a function, are a familiar idiom in functional programming. For example, the following two tail-recursive functions of type "`int -> bool`" determine whether a natural number is even or odd:

```
fun evenp n
    = if zerop n then true else oddp (sub1 n)
and oddp n
    = if zerop n then false else evenp (sub1 n)
```

where the recursive calls to `oddp` and `evenp` are shaded.

These functions follow the inductive structure of Peano numbers, they are mutually recursive (i.e., they may call each other), and all their calls are tail calls. Therefore they are tail-recursive. (N.B. The parity of a natural number is more quickly established with a modulus operation, but the point of this example is to illustrate tail recursion.)

The following shaded trace visualizes the computation determining whether 2 is even:

```
evenp 2
|if zerop 2 then true else oddp (sub1 2)
||zerop 2
||false
|if false then true else oddp (sub1 2)
|oddp (sub1 2)
||sub1 2
||1
|oddp 1
||if zerop 1 then false else evenp (sub1 1)
|||zerop 1
|||false
||if false then false else evenp (sub1 1)
||evenp (sub1 1)
|||sub1 1
|||0
||evenp 0
|||if zerop 0 then true else oddp (sub1 0)
||||zerop 0
||||true
|||if true then true else oddp (sub1 0)
|||true
||true
|true
true
```

This trace is naive because the number of bars on one line depends on the input, even though this dependency is spurious since each of the function calls is a tail call: it is the last thing that is done in the current context, and therefore there is no need to save this context and finish the computation with a cascade of function returns [341]. In a tail-call-optimized implementation, the trace reads as follows:

```
evenp 2
if zerop 2 then true else oddp (sub1 2)
|zerop 2
|false
if false then true else oddp (sub1 2)
oddp (sub1 2)
|sub1 2
|1
oddp 1
if zerop 1 then false else evenp (sub1 1)
|zerop 1
|false
if false then false else evenp (sub1 1)
evenp (sub1 1)
|sub1 1
|0
evenp 0
if zerop 0 then true else oddp (sub1 0)
|zerop 0
|true
if true then true else oddp (sub1 0)
true
```

The number of vertical bars at the beginning of a line in the trace is bounded and independent of the input.

Tail-call optimization makes it possible for functional programmers to implement iterative algorithms with tail-recursive functions. In fact, all modern functional languages are expected to have properly tail-recursive implementations [77]. So `evenp` and `oddp` actually implement the transition functions of an automaton with two states.

1.1.3 A fast exponentiation function for natural numbers

There is a faster way to decompose a natural number than peeling off its increment: dividing it by 2. The corresponding inductive construction reads as follows:[1]

$$\begin{aligned} n &::= 0 \mid m \\ m &::= 2m \mid 2n + 1 \end{aligned}$$

The following identities match this inductive construction:

$$\begin{aligned} x^0 &= 1 \\ x^{2m} &= (x^m)^2 \\ x^{2n+1} &= (x^n)^2 \times x \end{aligned}$$

[1] The auxiliary non-terminal in the BNF ensures that each natural number is represented uniquely.

Based on these equations, and using the auxiliary functions above, one can define an exponentiation function `power1` that given an integer `x` and a natural number `n`, computes x^n by dividing the exponent by 2 until the base case is reached. The definition uses two more auxiliary functions: the first one squares a given integer, and the second one computes the quotient of a given integer by 2 and a boolean indicating whether this integer was even (so for example applying `divmod2` to 17 yields `(8, false)`).

```
fun sqr n                                                (* sqr : int -> int *)
    = mul (n, n)

fun divmod2 n                                  (* divmod2 : int -> int * bool *)
    = (n div 2, evenp n)

fun power1 (x, n)                                (* power1 : int * int -> int *)
    = let fun visit n                                   (* visit : int -> int *)
              = if zerop n
                then 1
                else visit_positive n
          and visit_positive m                 (* visit_positive : int -> int *)
              = let val (q, r) = divmod2 m
                in if r
                   then sqr (visit_positive q)
                   else mul (sqr (visit q), x)
                end
      in visit n
      end
```

For example, raising 2 to the 10th power gives rise to the following trace:

```
power1 (2, 10)
visit 10
...
visit_positive 10
|...
|visit_positive 5
|...
||visit 2
||...
||visit_positive 2
||...
|||visit_positive 1
|||...
||||visit 0
||||...
||||1
|||...
|||2
||...
||4
|...
|32
...
1024
```

This definition of the power function is slightly unusual: in the else branch of `visit`, the information that `n` denotes a strictly positive natural number is usually ignored and `visit_positive` is inlined; or the base test is duplicated and `visit` is inlined. In either case, the power function uses one auxiliary function, not two. Nevertheless, having two local functions makes it possible to illustrate more concepts later on in this essay, in Sections 1.1.4, 1.2.1, 2.1, 2.4.2, and 2.4.3.

1.1.4 Generic programming

With their emphasis on abstraction, functional languages encourage programmers to think generically. For example, the inductive definition in Section 1.1.3 suggests the following 'fold' function that traverses a natural number by successively dividing it by 2; it is parameterized by a value for the base case and by two functions that determine what to do in the two induction cases:

```
(*  fold_nat2 : 'a * ('a -> 'a) * ('a -> 'a) -> int -> 'a  *)
fun fold_nat2 (base, even, odd) n
    = let fun visit n                                    (* visit : int -> 'a *)
              = if zerop n
                then base
                else visit_positive n
          and visit_positive m                (* visit_positive : int -> 'a *)
              = let val (q, r) = divmod2 m
                in if r
                   then even (visit_positive q)
                   else odd (visit q)
                end
      in visit n
      end
```

One can then define a fast exponentiation function as an instance of `fold_nat2` by supplying a value for the base case and two functions for the induction case:

```
(*  power1_with_fold_nat2 : int * int -> int  *)
fun power1_with_fold_nat2 (x, n)
    = fold_nat2 (1, sqr, fn v => mul (sqr v, x)) n
```

As a soundness check, one can calculate a less abstract definition of `power1_with_fold_nat2` by specializing `fold_nat2` with respect to its three parameters. The calculation consists of inlining the call to `fold_nat2` and then inlining the call to the second function (see Section 4 for more about partial evaluation). The result coincides with the definition of `power1`.

1.1.5 Divide and conquer for floating-point numbers

The strategy of dividing a number can also be applied to floating-point numbers, defining the base case as a degree of precision. For example, $\sin x$ can be approximated by x when x is sufficiently small. The following identity provides an inductive case:

$$\sin(3x) = 3\sin x - 4(\sin x)^3$$

One can then implement the sine function by successively dividing its argument by 3 until the degree of precision is reached, and then by successively returning an intermediate result, based on the approximation and the identity just above:

```
fun sin0 (x, epsilon)                          (* sin0 : real * real -> real *)
    = let fun visit x                                (* visit : real -> real *)
              = if abs x < epsilon
                then x
                else let val s = visit (x / 3.0)
                     in 3.0 * s - 4.0 * s * s * s
                     end
      in visit x
      end
```

This recursive program originates in Item 158 of HAKMEM [32]. It is revisited in Section 1.2.1, page 11 and in Section 2.2, pages 40 and 43.

1.1.6 Lazy data constructors

Given a list of n elements, the following program yields a list of n pairs, one for each element in the list together with the rest of the list, minus this element. For example, it maps the input list `[1,2,3]` to the output list `[(1,[2,3]),(2,[1,3]),(3,[1,2])]`.

The local function `visit` traverses the input list: at call time, it accumulates the reverse of its prefix; and at return time, (1) it prepends the current prefix to the current rest of the input list, using the auxiliary function `reverse_prepend`, and (2) it constructs the result:

```
fun reverse_prepend (nil, a)
    = a
  | reverse_prepend (x :: xs, a)
    = reverse_prepend (xs, x :: a)

fun enumerate xs               (* enumerate : 'a list -> ('a * 'a list) list *)
    = let fun visit (nil, p)
              = nil
            | visit (x :: xs, p)
              = (x, reverse_prepend (p, xs)) :: (visit (xs, x :: p))
      in visit (xs, nil)
      end
```

This program can be turned into one that uses a *generator*, which is one of the features of the Icon programming language [**125**, 197, 301]. To this end, the function producing the result is parameterized with lazy-list constructors [184]:

```
fun generate (xs, lazy_nil, lazy_cons)
    = let fun visit (nil, p)
              = lazy_nil ()
            | visit (x :: xs, p)
              = lazy_cons ((x, reverse_prepend (p, xs)),
                           fn () => visit (xs, x :: p))
      in visit (xs, nil)
      end

fun enumerate xs               (* enumerate : 'a list -> ('a * 'a list) list *)
    = generate (xs, fn () => nil, fn (r, t) => r :: t ())
```

The key function `generate` is given the input list and two parameters specifying how to construct the resulting list. The main function `enumerate` supplies `generate` with the input list and two constructors for the result. This example is revisited in Section 1.6.

1.2 Continuation-passing style (CPS)

In a CPS (continuation-passing style) program, no function ever returns: all calls are tail calls, all intermediate results are named, their computation is sequentialized, and in addition to their regular arguments, all functions are passed a continuation, i.e., a representation of "the rest of the computation" [347]. This functional representation of the rest of the computation is generally agreed to be the definition of a continuation [305] and it gave rise to the notion of 'continuation semantics' [190, 264, 317, 345]. The term "CPS" is due to Guy Steele [338].

As first pointed out by John Reynolds [304] and then formalized by Gordon Plotkin [295], CPS programs are insensitive to their evaluation order: evaluating them with a call-by-value strategy or with a call-by-name strategy yields the same result.

1.2.1 CPS with second-class continuations

Example: The fast exponentiation function for natural numbers. Let us assume the CPS counterparts of the elementary functions of Section 1.1:

```
(*  mul_c : int * int * (int -> 'a) -> 'a  *)
(*  sqr_c : int *       (int -> 'a) -> 'a  *)
```

In addition to their integer arguments, these two functions are passed a continuation that they apply to an intermediate result.[2] For `mul_c`, the given continuation is sent an integer which is the product of the two given integers. For `sqr_c`, the given continuation is sent an integer which is the square of the given integer. Their codomains are polymorphic to reflect that no assumption is made about the final answer [355].

Let us also assume the CPS counterpart of `zerop` and `divmod2`:

```
(*    zerop_c : int * (unit -> 'a) * (unit -> 'a) -> 'a  *)
(*  divmod2_c : int *  (int -> 'a) *  (int -> 'a) -> 'a  *)
```

Each of these functions is given two continuations. The first function applies one or the other to the unit value, depending on whether the given integer is `0`. The second function applies one or the other to the quotient of the given integer by 2, depending on whether this given integer is even or odd. The codomains of these two functions are also polymorphic.

Given an integer `x`, a natural number `n`, and a continuation `k`, the following CPS exponentiation function computes x^n and sends the result to `k`:

```
(*  power2 : int * int * (int -> 'a) -> 'a  *)
fun power2 (x, n, k)
    = let fun visit_c (n, k)
              = zerop_c (n,
                         fn () => k 1,
                         fn () => visit_positive_c (n, k))
          and visit_positive_c (m, k)
              = divmod2_c (m,
                           fn q => visit_positive_c
                                     (q, fn v => sqr_c
                                                 (v, k)),
```

[2] Since Carl Hewitt's work on Actors [214], "applying the continuation to an intermediate result" is often referred to as "sending this intermediate result to the continuation".

```
                        fn q => visit_c
                                (q, fn v => sqr_c
                                            (v, fn v2 => mul_c
                                                         (v2, x, k))))
      in visit_c (n, k)
      end
```

The following trace illustrates that all calls are tail calls:

```
power2 (2, 10, fn a => ...)
visit_c (10, fn a => ...)
zerop_c (10, fn () => ..., fn () => visit_positive_c (10, fn a => ...))
(fn () => visit_positive_c (10, fn a => ...)) ()
visit_positive_c (10, fn a => ...)
divmod_c (10, fn q => visit_positive_c (q, ...), fn q => visit_c (q, ...))
(fn q => visit_positive_c (q, ...)) 5
visit_positive_c (5, ...)
...
(fn a => ...) 1024
```

There are no nested calls and so no vertical bars at the beginning of any line in the trace.

Mixing direct and continuation-passing styles: In programming practice, not all functions are in CPS: the predefined ones are typically left in direct style.[3] For example, here is the fast exponentiation function in this mixed style:

```
(*  power3 : int * int * (int -> 'a) -> 'a  *)
fun power3 (x, n, k)
    = let fun visit_c (n, k)
              = if zerop n
                then k 1
                else visit_positive_c (n, k)
          and visit_positive_c (m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive_c (q, fn v => k (sqr v))
                   else visit_c (q, fn v => k (mul (sqr v, x)))
                end
      in visit_c (n, k)
      end
```

The following trace illustrates the computation of 2 to the 10th power:

```
power3 (2, 10, fn a => a)
visit_c (10, fn a => a)
if zerop 10 then (fn a => a) 1 else visit_positive_c (10, fn a => a)
|(zerop 10)
|false
if false then (fn a => a) 1 else visit_positive_c (10, fn a => a)
visit_positive_c (10, fn a => a)
let val (q, r) = divmod2 10 in if r then ... else ... end
|divmod2 10
```

[3] A mixed CPS program is evaluation-order independent if its direct-style functions are pure and total [127].

```
|(5, true)
let val (q, r) = (5, true) in if r then ... else ... end
if true then visit_positive_c (5, ...) else visit_c (5, ...)
visit_positive_c (5, ...)
...
(fn a => a) 1024
```

The maximum number of nested calls (i.e., of vertical bars at the beginning of a line in the trace) is bounded and independent of the input.

Interfacing the direct-style world and the CPS world:

- A function that is not in CPS is interfaced with a function that is in CPS by providing an initial continuation. For example, one can define the sine function from Section 1.1.5 with a direct-style interface but still with a local CPS function as follows:

```
fun sin1 (x, epsilon)
    = let fun visit_c (x, k)
              = if abs x < epsilon
                then k x
                else visit_c (x / 3.0,
                              fn s => k (3.0 * s - 4.0 * s * s * s))
      in visit_c (x, fn s => s)
      end
```

  Such a definition can be obtained by what John Reppy refers to as a 'local CPS conversion' [303] (see Section 2.3).

- Conversely, a CPS function is interfaced with a direct-style function using a non-tail call to the direct-style function and sending the result to the continuation. For example, `sqr_c` can be defined either by calling `sqr` or by inlining its call:

```
fun sqr_c (n, k)
    = k (sqr n)
(*  = k (mul (n, n))  *)
```

  The call to `sqr`, in the former case, and the call to `mul`, in the latter case, are not in tail position; `power3` is written in such a mixed style above.

Interfacing the direct-style world and the CPS world prefigures control delimiters (see Section 1.4).

Generic programming: The fold function for natural numbers can also be instantiated with continuation-passing arguments to define a fast exponentiation function:

```
fun power3_with_fold_nat2 (x, n, k)
    = fold_nat2 (fn k => k 1,
                 fn c => fn k => c (fn v => k (sqr v)),
                 fn c => fn k => c (fn v => k (mul (sqr v, x))))
                n
                k
```

This function behaves analogously to `power3`. Furthermore, as for `power1_with_fold_nat2` in Section 1.1.4, one can obtain not just the behavior of `power3` but a curried version of its actual code by specializing `fold_nat2` with respect to its three parameters.

Occurrence condition: In addition to their structural specificities (all calls being tail calls, etc.), the CPS programs corresponding to direct-style programs have a remarkable syntactic property—one identifier is enough to denote the current continuation [**110**, 141].

1.2.2 CPS with first-class continuations

A common mistake for CPS beginners is to forget to apply the current continuation when reaching an intermediate result. Their program therefore stops with this intermediate result. This common mistake, however, illustrates how to stop the computation at any point.

Stopping the computation: Consider the case of dividing by `0`. The following continuation-passing function is given two natural numbers `i` and `j` and a continuation `k`; if `j` is not `0`, it continues the computation by sending the quotient and the remainder of `i` and `j` to `k`; otherwise, it stops the computation with an error message. The final result of the program (traditionally referred to as the 'answer') is therefore a sum:

```
datatype 'a answer = VALUE of 'a
                   | ERROR of string

(*  divmod_c : int * int * (int * int -> 'a answer) -> 'a answer  *)
fun divmod_c (i, 0, k)
    = ERROR ("division of " ^ Int.toString i ^ " by 0")
  | divmod_c (i, j, k)
    = k (i div j, i mod j)
```

In the first clause, the current continuation is not applied and the computation stops. In the second clause, the current continuation is applied and the computation continues.

Continuing the computation elsewhere: The chief reason for using CPS in programming is to relax the occurrence condition mentioned above, and allow the use of *another* continuation than the current one.[4] The classical example of Calder mobiles illustrates this point. A Calder mobile is either an object (of a certain weight) or a bar (of a certain weight) holding two Calder sub-mobiles:

```
datatype mobile = OBJ of int
                | BAR of int * mobile * mobile
```

For example, `m1` and `m2` below denote two mobiles:

```
val m1 = BAR (1, BAR (1, OBJ 2, OBJ 2), OBJ 5)
val m2 = BAR (1, OBJ 6, BAR (1, OBJ 2, OBJ 9))
```

The first is well-balanced in the sense that the two sub-mobiles of each branch have the same weight. The second is not, because of "`BAR (1, OBJ 2, OBJ 9)`".

The challenge is to test the balance of a given mobile by traversing it *at most once*. As proposed by Patrick Greussay [195, page 67], one could (1) use continuations to traverse the given mobile, (2) incrementally compute its weight in passing, and (3) only apply the current continuation if the current sub-mobile is balanced. If the current sub-mobile is unbalanced, the current continuation is not applied; instead, the continuation that was given together with the input mobile is applied to `false`. If all the sub-mobiles are balanced, the continuation that was given together with the input mobile is eventually applied to `true`.

[4] A continuation that is invoked out of turn, i.e., that is invoked instead of the current continuation, is said to be "first class", and the others are said to be "second class" [**110**, 130]. These terms are due to Christopher Strachey [346] for functions and to Daniel Friedman and Christopher Haynes for continuations [180].

```
(*  bal1 : mobile * (bool -> 'a) -> 'a  *)
fun bal1 (m, k')
    = let (*  visit_c : mobile * (int -> 'a) -> 'a  *)
          fun visit_c (OBJ n, k)
              = k n
            | visit_c (BAR (n, m1, m2), k)
              = visit_c (m1, fn n1 => visit_c (m2, fn n2 => if n1 = n2
                                                            then k (n + n1 + n2)
                                                            else k' false))
      in visit_c (m, fn _ => k' true)
      end
```

Summary: As illustrated just above, in CPS,

- the current continuation is invoked explicitly to continue the computation with an intermediate result;
- invoking a continuation other than the current one has the effect of transferring control to this other continuation, never to return to the current one, as in a jump; and
- not invoking any continuation at all has the effect of stopping the computation.

The price to pay for this extra expressive power is CPS itself: complete programs need to be written in CPS, even the parts with second-class continuations only.

1.2.3 Undelimited first-class continuations in direct style

One programming technique for handling conditions
was by disjoint-union error indications, delivered level by level.
But the violence this did to our programs
was an argument for hiding it in the interpretive mechanism.
Peter J. Landin, 1997 [247]

Which violence did Peter Landin refer to when he wrote these lines? To illustrate his argument, let us go back to the Calder mobiles and use an `option` type. The local function `visit` maps an unbalanced sub-mobile to "`NONE`" and a balanced sub-mobile to "`SOME n`" where `n` denotes the weight of this sub-mobile:

```
(*  bal2 : mobile -> bool  *)
fun bal2 m
    = let (*  visit : mobile -> int option  *)
          fun visit (OBJ n)
              = SOME n
            | visit (BAR (n, m1, m2))
              = (case visit m1
                   of NONE    => NONE
                    | SOME n1 => (case visit m2
                                    of NONE    => NONE
                                     | SOME n2 => if n1 = n2
                                                  then SOME (n + n1 + n2)
                                                  else NONE))
      in case visit m
           of NONE   => false
            | SOME n => true
      end
```

Unlike that of `bal1`, this definition is cluttered by constructing optional values and dispatching over them in cascade. Its CPS counterpart does not directly correspond to the definition of `bal1` either, since the type of `visit_c` is "`mobile * (int option -> 'a) -> 'a`" instead of "`mobile * (int -> 'a) -> 'a`" which differs in the domain of the continuation. To obtain the definition of `bal1`, one can split the continuation "`int option -> 'a`" into "`(int -> 'a) * (unit -> 'a)`". The first continuation is applied to the weight of the current sub-mobile if it is balanced and the second to the unit value if the current sub-mobile is unbalanced. Inlining the second continuation yields the definition of `bal1` [116].

How could one realize this simplification in direct style rather than only in CPS? Landin suggested to use a *control operator* to this end [245]. The control operator would take a snapshot of the current continuation [277] and make it possible to restore this continuation whenever wanted.

> *The ominous list [of deviations from strict lambda] would not grow longer if,*
> *to the familiar idea of non-local jumps,*
> *was added the less familiar but pressing idea of jumps*
> *that had arguments and results and did some calculating in between.*
> Peter J. Landin, 1997 [247]

Landin originally proposed a control operator, J, that proved a remarkably good fit to model Algol's labels and jumps [162, Section 1] [244]. Since this proposal, Reynolds's escape operator [304] and call/cc in Scheme [73, 235] and in Standard ML of New Jersey [15, 157] have been deemed to be more convenient for general-purpose programming [196]. For example, the direct-style counterpart of `bal1` is written as follows, using `callcc` and `throw` as found in the library `SMLofNJ.Cont` of Standard ML of New Jersey, that respectively capture the current continuation and restore a captured continuation:

```
(*  type 'a cont                          *)
(*  val callcc : ('a cont -> 'a) -> 'a  *)
(*  val throw : 'a cont -> 'a -> 'b     *)

fun bal3 m
    = callcc (fn k' => let (*  visit : mobile -> int  *)
                           fun visit (OBJ n)
                               = n
                             | visit (BAR (n, m1, m2))
                               = let val n1 = visit m1
                                     val n2 = visit m2
                                 in if n1 = n2
                                    then n + n1 + n2
                                    else throw k' false
                                 end
                       in let val _ = visit m
                          in true
                          end
                       end)
```

When `bal3` is given a mobile, its implicit continuation is captured and bound to `k'`. This continuation is explicitly resumed if one of the sub-mobiles is unbalanced; otherwise, it is implicitly resumed when the computation completes. Landin's point is that the definition of `bal3` with its non-local jump is simpler to write than the definition of `bal2`.

Other control operators:

Exceptions: The non-local jump in `bal3` could be implemented with an exception instead of `callcc`. First-class continuations, however, are less constrained than non-local jumps and exceptions and they can be used, independently of any (control-) stack discipline, to concisely implement, e.g., process queues [62] and operating-system services [366].

Abortion: Stopping the computation is customarily done with an `abort` operator. For example, the `divmod_` example of Section 1.2.2, page 12 where the current continuation is tossed away, is written in direct style as:

```
fun divmod (i, 0)
    = abort ("division of " ^ Int.toString i ^ " by 0")
  | divmod (i, j)
    = (i div j, i mod j)
```

Felleisen's $\mathcal{C}$ operator: Together, `callcc` and `abort` make it possible to define $\mathcal{C}$ [161]:

```
(*  C : ('a cont -> 'b) -> void  *)
fun C f
    = callcc (fn k => abort (f k))
```

Whereas `callcc` takes a snapshot of the current continuation while leaving it in place, $\mathcal{C}$ grabs the current continuation and leaves nothing in place, which is convenient, e.g., for simulating coroutines. To resume the computation at the point of capture, the programmer must explicitly invoke the captured continuation. So for example, using $\mathcal{C}$ instead of `callcc` in the definition of `bal3` would require the following adjustment:

```
fun bal4 m
    = C (fn k' => let fun visit (OBJ n)
                          = n
                        | visit (BAR (n, m1, m2))
                          = let val n1 = visit m1
                                val n2 = visit m2
                            in if n1 = n2
                               then n + n1 + n2
                               else throw k' false
                            end
                  in let val _ = visit m
                     in throw k' true
                     end
                  end)
```

Summary: As illustrated just above, in direct style,

- the current continuation is invoked implicitly to continue the computation with an intermediate result;
- invoking a captured continuation has the effect of abandoning the current one and transferring control to this other continuation, never to return to the current one, as in a jump; and
- `abort` has the effect of stopping the computation.

1.3 Beyond CPS

The common mistake mentioned in Section 1.2.2 (forgetting to apply a continuation and thus stopping the computation with an intermediate result) is promptly noticeable, even for a beginner. Considerably less noticeable is the bug of making non-tail function calls in a "CPS" program. This bug leads the beginner to providing initial continuations left and right as the need arises and often makes his program evaluation-order dependent.

The goal of this section is to show that this beginner's bug may be turned into an advanced feature: in such non-tail-recursive programs, initializing the continuation naturally *delimits* it. Such a delimited continuation does not represent the rest of the computation, but only a prefix of it, up to this initialization. Therefore:

- not applying the current delimited continuation has the effect of resuming the computation at the delimitation point; and
- applying a delimited continuation may return a result at the point of call, as if the delimited continuation were a normal function; in other words, a delimited continuation can be *called* whereas an undelimited continuation can only be jumped to.

The rest of this section illustrates these two points with two direct-style programs that call a function expecting a continuation. This function is passed an initial continuation—typically the identity function. Therefore the continuation of this function does *not* represent the entire rest of the computation, but only a *delimited* part of it, up to the point of call of this function. If this function terminates, it will return a result at its point of call, no matter what it does with its continuation, e.g., not applying it at all (Section 1.3.1) or applying it out of turn (Section 1.3.2). These two programs go beyond CPS (Section 1.3.3).

1.3.1 Releasing control

For example, the function testing whether a mobile is balanced could itself be in direct style and have a local function in CPS. In this local function,

- if all the sub-mobiles are balanced, the initial continuation is eventually applied (to the total weight of the mobile) and `true` is returned to the caller of `bal5`, and
- if the current sub-mobile is unbalanced, the current continuation is not applied; instead, `false` is returned to the caller of `bal5`.

```
fun bal5 m
    = let (*  visit_c : mobile * (int -> bool) -> bool  *)
          fun visit_c (OBJ n, k)
              = k n
            | visit_c (BAR (n, m1, m2), k)
              = visit_c (m1, fn n1 => visit_c (m2, fn n2 => if n1 = n2
                                                            then k (n + n1 + n2)
                                                            else false))
      in visit_c (m, fn _ => true)
      end
```

1.3.2 Regaining control

To symbolically distribute products over sums in an arithmetic expression, one is given:

- a source data type of expressions with mixed sums and products, for some type `ide`:

```
datatype expression = IDE of ide
                    | ADD of expression * expression
                    | MUL of expression * expression
```

- and a target data type of sums of products:

```
datatype product = IDE_PROD of ide
                 | MUL_PROD of product * product

datatype sum_of_products = PROD of product
                         | ADD_PROD of sum_of_products * sum_of_products
```

The following continuation-based program implements such a mapping:

```
(*  distribute : expression -> sum_of_products  *)
fun distribute e
    = let (*  expression * (product -> sum_of_products) -> sum_of_products  *)
          fun visit_c (IDE x, k)
              = k (IDE_PROD x)
            | visit_c (ADD (e1, e2), k)
              = ADD_PROD (visit_c (e1, k), visit_c (e2, k))
            | visit_c (MUL (e1, e2), k)
              = visit_c (e1,
                         fn p1 => visit_c (e2,
                                           fn p2 => k (MUL_PROD (p1, p2))))
      in visit_c (e, fn p => PROD p)
      end
```

This program transgresses CPS in three ways:

- the initial continuation interfaces `visit_c` and `distribute`;
- in the clause for `ADD`, the current continuation is duplicated, and therefore the current call to `visit_c` will "return twice" at run time; and
- in the clause for `ADD`, both calls to `visit_c` are not tail calls.

1.3.3 Beyond CPS, yes, but how much really?

To summarize, an ML function such as "`fn (f, k) => k (f (12345, fn v => v))`" is

- either a beginner's bug that should be corrected as "`fn (f, k) => f (12345, k)`" since all calls should be tail calls in CPS,
- or an advanced feature relative to delimiting the continuation of the call to `f`, which actually is at the root of the functional approach to backtracking [1, 64, 69, 258, 315, 350, 363], an approach that is currently enjoying a renewal of interest [**46**, 122, **125**, 218, 238, 335, 374] and is illustrated in Section 1.6.

So assuming that one is intent on delimiting control for backtracking, does this all mean that CPS falls short? It is the point of the next section that on the contrary, thinking so would make one fall short of CPS: a program where not all calls are tail calls is not in CPS, but still it can be expressed in CPS, as developed in the next section.

1.4 Continuation-passing style with two layers of continuations (2CPS)

The CPS counterpart of the identity function reads as "`fn (x, k) => k x`", and its CPS counterpart reads as "`fn (x, k, mk) => k (x, mk)`". Any CPS program can be further written in CPS by adding one layer of continuations. The original (and now inner) continuation takes the outer continuation as parameter. These programs are said to be in "2CPS". The outer continuation is usually referred to as the "meta-continuation".

These continuations have continuations.
Peter J. Landin, 1997 [247]

1.4.1 2CPS with second-class continuations

Going back to the example of Section 1.3.3, the CPS counterpart of the ML function

```
fn (f, k) => k (f (12345, fn v => v))
```

reads

```
fn (f, k, mk) => f (12345, fn (v, mk) => mk v, fn v => k (v, mk))
```

where all calls are tail calls. Examples of 2CPS programs most notably include programs that implement backtracking with a 'success' and a 'failure' continuation: the success continuation is the delimited continuation and the failure continuation is the meta-continuation.

The paradox that continuations have continuations is resolved by observing that the attribute of being a continuation is contextual, i.e., positional, not innate.
Peter J. Landin, 1997 [247]

Occurrence conditions: Just as in the last paragraph of Section 1.2.1, it can be observed that for the 2CPS programs corresponding to direct-style programs, two identifiers are enough for the two continuations. These identifiers respectively denote the current delimited continuation and the current meta-continuation.

1.4.2 2CPS with first-class delimited continuations

As in Section 1.2.2, relaxing the occurrence conditions over delimited-continuation identifiers gives rise to first-class delimited continuations:

- the current delimited continuation is invoked explicitly to continue the computation with an intermediate result and the current meta-continuation;
- invoking a delimited continuation other than the current one (and passing it the current meta-continuation) has the effect of transferring control to this other delimited continuation, never to return to the current one, as in a jump; and
- not invoking any delimited continuation (but invoking the current meta-continuation) has the effect of stopping the current delimited computation and continuing the computation beyond the current delimitation.

Together, the current delimited continuation and the current meta-continuation define the rest of the computation, and so one can still obtain the effect of `callcc`, `abort`, or $\mathcal{C}$. In addition though, and as developed next, one can specify their delimited analogues.

1.4.3 Delimited first-class continuations in direct style

In direct style, the current continuation is initialized with a "control delimiter" [163] such as `reset` [122]. For example, the direct-style counterpart of the ML function

```
fn (f, k) => k (f (12345, fn v => v))
```

from Section 1.4.1 reads

```
fn f => reset (fn () => f 12345)
```

For the rest, it is simple to adapt the previous control operators `callcc`, `abort`, and $\mathcal{C}$ to operate over the current delimited continuation instead of the current undelimited continuation. For example, the direct-style counterpart of `bal5` on page 16 is written as follows, using the control delimiter `reset : (unit -> bool) -> bool` and the delimited abort operator `abort1 : bool -> int`:

```
fun bal6 m
    = let fun visit (OBJ n)
              = n
            | visit (BAR (n, m1, m2))
              = let val n1 = visit m1
                    val n2 = visit m2
                in if n1 = n2
                   then n + n1 + n2
                   else abort1 false
                end
      in reset (fn () => let val _ = visit m in true end)
      end
```

1.4.4 From jumpy to pushy delimited continuations

The point of the functional approach to backtracking is not only to jump: it is to use a delimited continuation—e.g., the success continuation—as a function: a value whose call returns a result at its point of call. With 2CPS, the infrastructure is already there to support this behavior without compromising tail-recursion: when a delimited continuation is to be applied, the current delimited continuation can be "pushed" on the meta-continuation to be resumed later on. From here on, a delimited continuation is said to be "jumpy" if it abandons the current delimited continuation of its resumption, and it is said to be "pushy" if it pushes this current delimited continuation on the current meta-continuation.

Let us illustrate first-class delimited continuations in direct style with two examples: a toy one and the direct-style counterpart of the example of Section 1.3.2.

Example: relocating a context using pushy continuations: In this example, the current continuation is delimited with `reset`, a delimited continuation is captured with `shift`, and the captured continuation is called like a function (instead of being jumped to as with call/cc).

```
1 + reset (fn () => 10 + shift (fn k => k 100 + 1000))
```

The control delimiter isolates the outer addition and `k` is made to denote "`fn v => 10 + v`". Evaluating this expression is therefore akin to evaluating

```
1 + (let val k = fn v => 10 + v in k 100 + 1000 end)
```

and it yields `1111`.

The CPS counterpart of this expression reads

```
fn k' => let val k'' = fn v => k' (1 + v)
         in k'' (let val k = fn v => 10 + v in k 100 + 1000 end)
         end
```

where a continuation (`k''`) is applied to a non-trivial term (`let val k = ...`) and where a continuation (`k`) is called non-tail-recursively.

A first-class delimited continuation can be composed with another one, as in the following variant where `k` is applied three times:

```
1 + reset (fn () => 10 + shift (fn k => k (k (k 100)) + 1000))
```

Evaluating this variant is akin to evaluating

```
1 + let val k = fn v => 10 + v in k (k (k 100)) + 1000 end
```

and it yields `1131`.

Example: distributing products over sums. The direct-style counterpart of the example of Section 1.3.2, page 17, uses `shift` and `reset`, and reads as follows:

```
(*  distribute_ds : expression -> sum_of_products  *)
fun distribute_ds e
    = let (*  visit : expression -> product  *)
          fun visit (IDE x)
              = IDE_PROD x
            | visit (ADD (e1, e2))
              = shift (fn k => ADD_PROD (reset (fn () => k (visit e1)),
                                         reset (fn () => k (visit e2))))
            | visit (MUL (e1, e2))
              = MUL_PROD (visit e1, visit e2)
      in reset (fn () => PROD (visit e))
      end
```

where

- `shift : ((product -> sum_of_products) -> sum_of_products) -> product`
- `reset : (unit -> sum_of_products) -> sum_of_products`

As in the original definition of `distribute` on page 17, `k` has type `product -> sum_of_products`.

The three ways the definition of `distribute` transgressed CPS on page 17 are reflected in direct style as follows:

- the initial call to `visit` is delimited by `reset`;
- in the clause for `ADD`, the delimited current continuation is captured and called twice; and
- in the clause for `ADD`, both of the resumption contexts for `visit` are delimited.

Delimited control operators in ML: All the examples presented here use Andrzej Filinski's implementation of `shift` and `reset` in Standard ML of New Jersey [169]. This implementation takes the form of an ML functor mapping a type of intermediate answers to an ML structure satisfying the following signature:

```
signature SHIFT_AND_RESET
= sig
    type intermediate_answer
    val shift : (('a -> intermediate_answer) -> intermediate_answer) -> 'a
    val reset : (unit -> intermediate_answer) -> intermediate_answer
  end
```

So for example, defining an instance of `shift` and `reset` with an integer type of intermediate answers is done as follows:

```
structure R = Shift_and_Reset (type intermediate_answer = int)
```

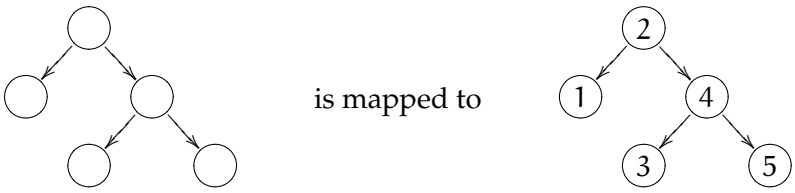
The toy example at the beginning of the section therefore actually reads:

```
1 + R.reset (fn () => 10 + R.shift (fn k => k 100 + 1000))
```

Similarly, `shift` and `reset` in the second example were defined as follows:

```
structure R = Shift_and_Reset (type intermediate_answer = sum_of_products)
```

Example: Simulating state with delimited control. Suppose one wants to label a binary tree with successive integers in a depth-first and infix way, so that a tree such as



A natural way to proceed is to traverse the given tree recursively, threading a counter:

```
datatype tree = LEAF of int
              | NODE of tree * int * tree

fun label_infix0 t                              (* label_infix0 : tree -> tree *)
    = let fun inc i                                  (* inc : int -> int * int *)
              = (i, i+1)
          fun visit (LEAF _, i)          (* visit : tree * int -> tree * int *)
              = (LEAF i, i+1)
            | visit (NODE (t1, _, t2), i)
              = let val (t1', i1) = visit (t1, i)
                    val (j, i1') = inc i1
                    val (t2', i2) = visit (t2, i1')
                in (NODE (t1', j, t2'), i2)
                end
      in #1 (visit (t, 1))
      end
```

The counter behaves as a state. In the CPS version of this program, this state can be eta-reduced away in the definition of `visit`, and only accessed through the auxiliary function `inc`, using the type "`int -> tree`" for intermediate answers:

```
fun label_infix1 t                              (* label_infix1 : tree -> tree *)
    = let fun inc ((), k)          (* inc : unit * (int -> answer) -> answer *)
              = fn i => k i (i+1)
          fun visit (LEAF _, k) (* visit : tree * (tree -> answer) -> answer *)
              = inc ((), fn i => k (LEAF i))
            | visit (NODE (t1, _, t2), k)
              = visit (t1, fn t1' =>
                  inc ((), fn j =>
                   visit (t2, fn t2' =>
                     k (NODE (t1', j, t2')))))
      in visit (t, fn t' => fn i => t') 1
      end
```

This program can be expressed in direct style using `shift` and `reset`:

```
structure S = Shift_and_Reset (type intermediate_answer = int -> tree)

fun label_infix2 t                              (* label_infix2 : tree -> tree *)
    = let fun inc ()                                      (* inc : unit -> int *)
              = S.shift (fn k => fn i => k i (i+1))
          fun visit (LEAF _)                          (* visit : tree -> tree *)
              = LEAF (inc ())
            | visit (NODE (t1, _, t2))
              = NODE (visit t1, inc (), visit t2)
      in S.reset (fn () => let val t' = visit t in fn i' => t' end) 1
      end
```

To use Landin's words from Section 1.2.3., the definition of `visit` incurs less violence since the counter is hidden in the background. This use of a side effect is very disciplined since `label_infix1` is the CPS counterpart of `label_infix2`.

This example suggests a simple simulation of state where an integer is threaded in the background, read with a function `get`, and modified with a function `set`, assuming one wants to compute, e.g., a list of integers:

```
structure S = Shift_and_Reset (type intermediate_answer = int -> int list)
fun get () = S.shift (fn k => fn i => k i i)
fun set i' = S.shift (fn k => fn i => k () i')
fun with_int_state (i, t)
    = S.reset (fn () => let val result = t () in fn i' => result end) i
```

Evaluating the following expression successively reads and writes the state in a way reminiscent of using a state monad [170]:

```
with_int_state (1, fn () => let val x1 = get ()
                                val () = set 10
                                val x2 = get ()
                                val () = set 100
                            in [x1, x2, get ()]
                            end)
```

The result is `[1,10,100]`, and to go back to Landin's opening words on page 2, the withering away of the statement is a phenomenon we might not quite live to see after all.

1.5 Continuation-passing style with n layers of continuations (nCPS): the CPS hierarchy

Too much of a good thing can be wonderful.
Mae West

Why stop at two layers of continuations? One can add arbitrarily many and obtain a CPS hierarchy [**41**, 122, 150, 171, 233]. By default, for n layers, n continuation identifiers are enough; they denote second-class continuations: $n-1$ are delimited and the last one is undelimited. Relaxing this occurrence condition makes it possible to implement, e.g., nested backtracking through first-class delimited continuations. The previous control operators can be generalized to more layers of continuations, or new operators can be designed as the need arises.

Delimited control operators in ML: Filinski has extended his implementation of `shift` and `reset` to the CPS hierarchy [171]. All the examples presented here use Yang's alternative implementation [150]. This implementation takes the form of an ML functor mapping a type of intermediate answers and a previously defined ML structure satisfying `SHIFT_AND_RESET` into a new ML structure that also satisfies `SHIFT_AND_RESET`.

For example, defining an instance of `shift` and `reset` with a boolean type of intermediate answers over an instance of `shift` and `reset` with an integer type of intermediate answers is done as follows:

```
structure C1 = Shift_and_Reset (type intermediate_answer = int)

structure C2 = Shift_and_Reset_next (type intermediate_answer = bool
                                     structure Over = C1)
```

With this layering, the CPS counterparts of an ML function and its type would successively evolve as follows. The body of each successive version of the function (i.e., `e0`, `e1`, `e2`, and `e3`) is not detailed; what matters is its header (1, 2, 3, and then 4 parameters) and its type:

	term	type
direct style	`fn x => e0`	`: b -> b'` where `b` and `b'` are base types, for simplicity
1CPS	`fn (x, k1) => e1`	`: b * c1 -> int` where `c1 = b' -> int`

2CPS	`fn (x, k1, k2) => e2`	`: b * c1 * c2 -> bool`
		where `c1 = b' * c2 -> bool`
		`c2 = int -> bool`

3CPS	`fn (x, k1, k2, k3) => e3`	`: b * c1 * c2 * c3 -> 'a`
		where `c1 = b' * c2 * c3 -> 'a`
		`c2 = int * c3 -> 'a`
		`c3 = bool -> 'a`

The `int` type, in the second row, arises because of the declaration of `C1` above. The `bool` type, in the third row, arises because of the declaration of `C2` above. In the fourth row, the polymorphic type variable arises because the domain of answers is unconstrained.

1.6 Three continuation-based solutions to a cryptarithmetic puzzle

This section illustrates delimited continuations in the CPS hierarchy with three successive programs solving the following cryptarithmetic puzzle:

> Place the digits 1 through 9 in the cells of a 3×3-square so that adding the number formed by the digits of the first row and the number formed by the digits of the second yields the number formed by the digits of the last row. Each digit can be used only once.

For example, $124 + 659 = 783$. There are 336 solutions, all of which use a carry for adding the ones or the tens.

Section 1.6.1: The first program is purely functional and in 2CPS. It uses both a success and a failure continuation.

Section 1.6.2: The second program is in 1CPS. It uses a success continuation. Its failure continuation is implemented with shift and reset. This second program is the direct-style counterpart of the first one.

Section 1.6.3: The third program is in 0CPS, i.e., in direct style. Its success continuation is implemented with shift and reset, and its failure continuation is implemented with shift_2 and reset_2. This third program is the direct-style counterpart of the second one.

Each implementation lazily generates all possible triples and then successively checks each of them. To this end, 9 instances of the generator of page 8 are nested, instantiating `lazy_cons` with a success continuation and `lazy_nil` with a failure continuation, to paraphrase Philip Wadler [363]:

One programmer's lazy-list constructors
is another programmer's success and failure continuations.

1. Given `[d1,d2,d3,d4,d5,d6,d7,d8,d9]`, the first instance successively yields `(d1,[d2,d3,d4,d5,d6,d7,d8,d9])`, ..., and `(d9,[d1,d2,d3,d4,d5,d6,d7,d8])`.
2. Given `[d2,d3,d4,d5,d6,d7,d8,d9]`, the second instance successively yields `(d2,[d3,d4,d5,d6,d7,d8,d9])`, ..., and `(d9,[d2,d3,d4,d5,d6,d7,d8])`.

   ...

8. Given `[d8,d9]`, the eighth instance successively yields `(d8,[d9])` and `(d9,[d8])`.
9. Given `[d9]`, the ninth instance yields `(d9,[])`.

For simplicity, no particular optimization is made, e.g., to reduce the search space or to minimize the number of arithmetic operations.

The generator and the solver are parametric in their domain of answers, of which two kinds are considered:

- the solution first encountered, if any, and
- the list of all possible solutions (one could also consider their number).

1.6.1 2CPS

In Figure 1, the computation is based on success and failure continuations: each of `generate` and `solve` is given a list of digits `ds`, a success continuation `sc`, and a failure continuation `fc`, and yields a final answer.

- The function `generate`, for each digit in the given list, applies the success continuation to this digit, the rest of this list, and the failure continuation, which is threaded like a state. (Reminder: `reverse_prepend` is defined in Section 1.1.6, page 8.)
- The function `solve` proceeds by nesting 9 instances of the generator and checks the resulting 9 digits. In case of success, the corresponding 3 integers are sent to the success continuation, together with the failure continuation; otherwise, the unit value is sent to the failure continuation.

```
fun generate (ds, sc, fc)
    = let fun visit (nil, a)
              = fc ()
            | visit (d :: ds, a)
              = sc (d, reverse_prepend (a, ds), fn () => visit (ds, d :: a))
      in visit (ds, nil)
      end

fun solve (ds, sc, fc)
    = generate (ds, fn (d1, ds, fc) =>
       generate (ds, fn (d2, ds, fc) =>
        generate (ds, fn (d3, ds, fc) =>
         generate (ds, fn (d4, ds, fc) =>
          generate (ds, fn (d5, ds, fc) =>
           generate (ds, fn (d6, ds, fc) =>
            generate (ds, fn (d7, ds, fc) =>
             generate (ds, fn (d8, ds, fc) =>
              generate (ds, fn (d9, ds, fc) =>
               let val x1 = 100*d1 + 10*d2 + d3
                   val x2 = 100*d4 + 10*d5 + d6
                   val x3 = 100*d7 + 10*d8 + d9
               in if x1 + x2 = x3
                  then sc ((x1, x2, x3), fc)
                  else fc ()
               end,
              fc),
             fc),
            fc),
           fc),
          fc),
         fc),
        fc),
       fc),
      fc)
```

Figure 1: Solving the puzzle with two layers of continuations

The first solution, if any: The result is an option type manifesting that there may or may not be a solution.

```
type result_first = (int * int * int) option

fun main_first ()                        (* main_first : unit -> result_first *)
    = solve ([1,2,3,4,5,6,7,8,9],
             fn (t, fc) => SOME t,
             fn () => NONE)
```

The function `main_first` supplies the initial success continuation and the initial failure continuation: if the initial success continuation is applied, there is a solution, which is injected as such in the result; if the failure continuation is applied, there is no solution, and the other summand of the result is provided.

All the solutions: The result is a list of solutions.

```
type result_all = (int * int * int) list

fun main_all ()                            (* main_all : unit -> result_all *)
    = solve ([1,2,3,4,5,6,7,8,9],
             fn (t, fc) => t :: fc (),
             fn () => nil)
```

The function `main_all` supplies the initial success continuation and the initial failure continuation: if the initial success continuation is applied, there is a solution, which is prepended to the current list of solutions; if the initial failure continuation is applied, there are no more solutions, and the empty list is returned.

N.B. In the definition of `main_all`, the call to the failure continuation is not a tail call and therefore this definition is not in CPS. To have a tail call, one more layer of continuations is necessary, yielding a program in 3CPS:

```
fun main_all' ()                           (* main_all' : unit -> result_all *)
    = solve ([1,2,3,4,5,6,7,8,9],
             fn (t, fc) => fn ec => fc () (fn ts => ec (t :: ts)),
             fn () => fn ec => ec nil)
            (fn ts => ts)
```

The number of solutions: The result is an integer and the program is a clone of the program just above.

```
type result_count = int

fun main_count ()                        (* main_count : unit -> result_count *)
    = solve ([1,2,3,4,5,6,7,8,9],
             fn (t, fc) => 1 + fc (),
             fn () => 0)

fun main_count' ()                      (* main_count' : unit -> result_count *)
    = solve ([1,2,3,4,5,6,7,8,9],
             fn (t, fc) => fn ec => fc () (fn n => ec (1 + n)),
             fn () => fn ec => ec 0)
            (fn n => n)
```

1.6.2 1CPS

This solution is the direct-style counterpart of the 2CPS solution (see Section 2.3 for more about the CPS transformation and the direct-style transformation). In Figure 2, the computation is only based on success continuations. Failure is implemented in direct style, using a delimited control operator.

- The function `generate`, for each digit in a given list, applies the success continuation to this digit and the rest of this list.
- The function `solve` proceeds by nesting 9 instances of the generator and checks the resulting 9 digits. In case of success, the corresponding 3 integers are sent to the success continuation; otherwise, the unit value is returned to the implicit failure continuation.

The nominal result of `solve` is of unit type, manifesting that the actual collection of solutions is happening behind the scenes as a computational effect: the failure continuation is now implicit. The computation pertaining to the failure continuation is now carried out through shift and reset.

```
fun generate (ds, sc)
    = let fun visit (nil, a)
              = ()
            | visit (d :: ds, a)
              = let val () = sc (d, reverse_prepend (a, ds))
                in visit (ds, d :: a)
                end
      in visit (ds, nil)
      end

fun solve (ds, sc)
    = generate (ds, fn (d1, ds) =>
       generate (ds, fn (d2, ds) =>
        generate (ds, fn (d3, ds) =>
         generate (ds, fn (d4, ds) =>
          generate (ds, fn (d5, ds) =>
           generate (ds, fn (d6, ds) =>
            generate (ds, fn (d7, ds) =>
             generate (ds, fn (d8, ds) =>
              generate (ds, fn (d9, ds) =>
               let val x1 = 100*d1 + 10*d2 + d3
                   val x2 = 100*d4 + 10*d5 + d6
                   val x3 = 100*d7 + 10*d8 + d9
               in if x1 + x2 = x3
                  then sc (x1, x2, x3)
                  else ()
               end)))))))))
```

Figure 2: Solving the puzzle with one layer of continuations

The first solution, if any: The result is an option type manifesting that there may or may not be a solution.

```
type result_first = (int * int * int) option

structure C = Shift_and_Reset (type intermediate_answer = result_first)
val shift = C.shift (* : (('a -> result_first) -> result_first) -> 'a *)
val reset = C.reset (* : (unit -> result_first) -> result_first       *)

fun main_first ()                        (* main_first : unit -> result_first *)
    = reset (fn () => let val () = solve ([1,2,3,4,5,6,7,8,9],
                                          fn t => shift (fn fc => SOME t))
                      in NONE
                      end)
```

The function `main_first` supplies the initial success continuation and initializes the failure continuation with `reset`: if this initial success continuation is applied, there is a solution, and the (implicit) failure continuation is abstracted and tossed away, while the solution is injected as such in the final answer. If the computation completes, no solution has been found and the result is `NONE`.

All the solutions: The result is a list of solutions.

```
type result_all = (int * int * int) list

structure C = Shift_and_Reset (type intermediate_answer = result_all)
val shift = C.shift (* : (('a -> result_all) -> result_all) -> 'a *)
val reset = C.reset (* : (unit -> result_all) -> result_all       *)

fun main_all ()                            (* main_all : unit -> result_all *)
    = reset (fn () => let val () = solve ([1,2,3,4,5,6,7,8,9],
                                          fn t => shift (fn fc => t :: fc ()))
                      in nil
                      end)
```

The function `main_all` supplies the initial success continuation and initializes the failure continuation with `reset`: if this initial success continuation is applied, there is a solution, which is prepended to the current list of solutions. The (implicit) initial failure continuation is applied when the computation of `solve` completes; there are then no more solutions to be found and the empty list is returned.

The number of solutions: Here and in the next section, the program counting the number of solutions is left as an exercise to the reader.

1.6.3 0CPS (i.e., direct-style)

This solution is the direct-style counterpart of the 1CPS solution (see Section 2.3 for more about the CPS and the direct-style transformations). In Figure 3, the computation is now in direct style:

- The function `generate` captures its success continuation with shift_1, and for each digit in a given list, it applies this continuation to this digit and the rest of this list.
- The function `solve` proceeds by nesting 9 instances of the generator and checks the resulting 9 digits. In case of success, the corresponding 3 integers are returned to the implicit success continuation; otherwise, the implicit success continuation is captured, tossed away, and the unit value is returned to the twice-implicit failure continuation.

The nominal result of `generate` is of pair type, manifesting that the actual generation of solutions is happening behind the scenes as a computational effect: the success continuation is now implicit too. The computation pertaining to the success continuation is now carried

```
structure C1 = Shift_and_Reset (type intermediate_answer = unit)
val shift1 = C1.shift (* : (('a -> unit) -> unit) -> 'a *)
val reset1 = C1.reset (* : (unit -> unit) -> unit       *)

fun generate ds
    = shift1 (fn sc => let fun visit (nil, a)
                               = ()
                             | visit (d :: ds, a)
                               = let val () = sc (d, reverse_prepend (a, ds))
                                 in visit (ds, d :: a)
                                 end
                       in visit (ds, nil)
                       end)

fun solve ds
    = let val (d1, ds) = generate ds
          val (d2, ds) = generate ds
          val (d3, ds) = generate ds
          val (d4, ds) = generate ds
          val (d5, ds) = generate ds
          val (d6, ds) = generate ds
          val (d7, ds) = generate ds
          val (d8, ds) = generate ds
          val (d9, ds) = generate ds
          val x1 = 100*d1 + 10*d2 + d3
          val x2 = 100*d4 + 10*d5 + d6
          val x3 = 100*d7 + 10*d8 + d9
      in if x1 + x2 = x3
         then (x1, x2, x3)
         else shift1 (fn sc => ())
      end
```

Figure 3: Solving the puzzle with zero layers of continuations, i.e., in direct style

out through shift_1 and reset_1, and the computation pertaining to the failure continuation is now carried out through shift_2 and reset_2.

The first solution, if any: The result is an option type manifesting that there may or may not be a solution.

```
type result_first = (int * int * int) option

structure C2 = Shift_and_Reset_next (type intermediate_answer = result_first
                                     structure Over = C1)
val shift2 = C2.shift (* : (('a -> result_first) -> result_first) -> 'a *)
val reset2 = C2.reset (* : (unit -> result_first) -> result_first       *)

fun main_first ()                        (* main_first : unit -> result_first *)
    = reset2
       (fn () => let val () = reset1
                              (fn () => let val t = solve [1,2,3,4,5,6,7,8,9]
                                        in shift2 (fn fc => SOME t)
                                        end)
                 in NONE
                 end)
```

The function `main_first` sets the direct-style stage for the initial success continuation: if this initial success continuation is applied, then there is a solution, and the (implicit) failure continuation is abstracted and tossed away, while the solution is injected as such in the final answer. If the computation completes, no solution has been found and the result is `NONE`.

All the solutions: The result is a list of solutions.

```
type result_all = (int * int * int) list

structure C2 = Shift_and_Reset_next (type intermediate_answer = result_all
                                     structure Over = C1)
val shift2 = C2.shift (* : (('a -> result_all) -> result_all) -> 'a *)
val reset2 = C2.reset (* : (unit -> result_all) -> result_all       *)

fun main_all ()                             (* main_all : unit -> result_all *)
    = reset2
       (fn () => let val () = reset1
                              (fn () => let val t = solve [1,2,3,4,5,6,7,8,9]
                                        in shift2 (fn fc => t :: fc ())
                                        end)
                 in nil
                 end)
```

The function `main_all` sets the direct-style stage for the initial success continuation: if this initial success continuation is applied, then there is a solution, which is prepended to the list of solutions obtained by applying the current failure continuation. The (implicit) initial failure continuation is applied when the computation of `solve` completes; the empty list of solutions is then returned.

1.7 Related work, contribution, impact, and future work

To conclude this section, here is a review of each of the areas touched upon so far.

Programming patterns: CPS is a persistent source of inspiration to functional programmers, and the author makes no exception. For example, the author has used CPS to bypass the need for dependent types to program a string-formatting function [**104**], similarly to Andrzej Filinski's [172] and Zhe Yang's [381] inductive programming technique for type-directed partial evaluation [106]. This `printf` program has become popular in the area of typed functional programming [68, 187, 212, 219, 269, 285, 300, 324, 377, 381] and has made its way into the implementation of Standard ML of New Jersey <`http://www.smlnj.org`> and of MLton <`http://mlton.org/Printf`>. In addition, Kurt Bond's implementation in OCaml is available at <`http://tkb.mpl.com/~tkb/software.html#AEN546`>.

Mayer Goldberg and the author proposed a new programming pattern—nicknamed "There and Back Again" (TABA)—where one data structure is traversed at call time and *another one* at return time [**124**]. This programming pattern provided the topic of an exercise in Peter Van Roy and Seif Haridi's recent textbook *Concepts, Techniques, and Models of Computer Programming* [311, Chapter 3, Exercise 16] and it has elicited further work at the University of Tokyo [272]. Its logical content, however, remains to be investigated [278, Section 9.1].

The CPS and direct-style transformations: Section 2.3 connects continuation-based programs, whether in direct style with control operators or in CPS, using two program transformations: a CPS transformation mapping direct-style programs to their CPS counterpart [295, 338], and its left inverse, a direct-style transformation [**100**, 130, 248]. This pair of transformations provides guidelines to design, reason about, and relate programs such as the ones above about Calder mobiles. In that spirit, Kevin Millikin and the author have recently presented an array of new simulations of Landin's J operator [**135**].

Occurrence conditions: While the occurrence of continuation identifiers is essential to detect control operations in CPS [**110**, 130, 248], keeping track of the occurrences of the parameters of continuations is also essential to map CPS programs back to direct style correctly [**100**, 249]. For example, the two terms

```
fn k => f (x, fn v1 => g (x, fn v2 => v1 (v2, k)))
fn k => g (x, fn v2 => f (x, fn v1 => v1 (v2, k)))
```

should not both be mapped to the direct-style term "`f x (g x)`". Instead, assuming left-to-right call by value, the second one should be mapped to

```
let val v2 = g x in f x v2 end
```

to make it manifest that `g` is applied before `f`.

Pfenning and the author formalized that a CPS transformation yields terms satisfying occurrence conditions such that a direct-style transformation is a left-inverse of the CPS transformation [141]. Dzafic, Pfenning, and the author reported a relational formulation of the CPS transformation that makes it possible to prove properties about its output [120]. Pfenning identified that the occurrence conditions were accounted for by Intuitionistic Non-Commutative Linear Logic, and together with Jeff Polakow, he explored the consequences of this correspondence not only for CPS and functional programming but also for logic programming [297, 298, 299].

Implementation issues: In his PhD thesis, the author proposed a simple implementation technique for first-class continuations [96]. This technique is now both accepted [76, 217] and formalized [**110**]. The formalization uses the syntactic proof techniques of the above-mentioned joint work with Pfenning [141].

Reflective towers: In the early 1980's, Brian Smith introduced the model of a tower of interpreters to account for computational reflection [331, 332]. Indeed if a program is run by an interpreter (which is itself run by another interpreter), the computational state of the program is defined by the data structures operated upon by this interpreter:

- reifying this computational state into processable data is naturally achieved by *replacing* the interpreter by a program to process the current data structures operated upon by the interpreter; this program is run by the interpreter of the interpreter;
- reflecting upon a new program is achieved by spawning a new interpreter to run this new program.

In Smith's model, all the interpreters are metacircular, i.e., their defining language is the same as the language they define, and furthermore the tower is infinite, to allow programs to reify during reification. In practice, however, three levels are sufficient to simulate a reflective tower: one for the program, one for the interpreter, and one for its (meta-)interpreter. When reifying, the program is lifted up to the level of the interpreter, the meta-interpreter becomes the interpreter, and a new interpreter runs the interpreter. When reflecting upon a program fragment, the fragment becomes the new program, the program becomes the new interpreter, and the interpreter becomes the new meta-interpreter. The model was implemented by Brian Smith and Jim des Rivières with 3-Lisp [153].[5]

[5] On the next planet, in a building, there was a computer room.

The little prince: *"Good evening."*
The programmer, glued to his computer terminal: *"Sh! I'm programming!"*
The little prince: *"What is programming?"*
The programmer, his eyes still on the screen: *"Mmmmh ... you're interrupting me! What are you doing here?"*
The little prince: *"What is programming?"*
The programmer, blinking and turning his head towards the little prince: *"It is to make a machine do something. You write a program specifying what to do and the machine runs it."*
The little prince: *"So what do you want the machine to do?"*
The programmer: *"To run my program. I am making a program to run my program, meta-circularly."*
And the programmer returned to his meta-circular program.

The little prince pondered.

The little prince: *"Why do you want a program to run itself since the machine already runs it?"*
The programmer, looking at him again and frowning: *"Ah. You're still there. It is because I am building a tower of computation."*
The little prince: *"How tall will your tower be?"*
The programmer: *"Infinite."*
The little prince looked up.
The programmer: *"But in practice one only needs three levels."*
The little prince looked down again.
The programmer, resuming his task with a concentrated appearance: *"And I am working on it."*

"What a strange person. Every time I ask him what he wants, he tells me what he is doing," the little prince said to himself. *"So he must always be doing what he wants,"* he thought continuing his journey: *"a programmer must be very happy."*

Such a model could not leave the hitch-hikers of the meta-universe indifferent,[6] and indeed it did not. In two bold, incisive articles, "Reification: Reflection without Metaphysics" [181] and "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower" [373], Daniel Friedman and Mitchell Wand presented a programming-language approach to reflective towers, implementing a reflective language, Brown, in Scheme by means of a 'meta-continuation' to account for the outer layers of a tower of interpreters. Karoline Malmkjær and the author pursued this study with another reflective language, Blond [132], and Kenichi Asai with yet another one, Black [19], that is geared towards compilation via partial evaluation [22]. In these simulations, reinstating a reified continuation can either discard the current continuation and jump, or it can first push the current continuation onto the meta-continuation. So 3-Lisp, Brown, Blond, and Black provide support both for jumpy and pushy delimited continuations [121, Section 6.3]. In addition, these simulations of a reflective tower are interactive and, as in Matthias Felleisen's work on control delimiters [163], continuations are delimited by the prompt of the current interactive toplevel loop.

Control and prompt: In the late 1980's, Felleisen introduced the concept of control delimiters as a linguistic feature to support equational reasoning about direct-style programs with first-class continuations [163]:

- the control operator `prompt` (often noted #) delimits control, and
- the control operator `control` (often noted $\mathcal{F}$) abstracts control into a delimited continuation that can be composed with other functions rather than only be jumped to as with call/cc.

Felleisen's introduction of the concept of control delimiter in direct style was made independently of functional backtracking, CPS, and reflective towers, even though Sitaram and Felleisen have since connected control delimiters and CPS [329].

Shift and reset: In the late 1980's, Andrzej Filinski and the author proposed two control operators, shift and reset, that provide in direct style the ability to delimit continuations and to abstract delimited continuations as in CPS [121, 122, 123]. These control operators have found applications in non-deterministic programming (i.e., backtracking) [**46**, 122, 238], code generation and partial evaluation [20, 21, 156, 210, 250, 357, 349], type-directed evaluation and normalization by evaluation [27, **103**, **107**, 158, 174, 200, 210], computational monads [170, 174], mobile computing [348], and in the definition of the algorithmic language Scheme [235, page 105]. They have recently been axiomatized by Yukiyoshi Kameyama and Masahito Hasegawa [234].

The original typing discipline of shift and reset corresponds to the typing discipline of the CPS idioms they provide in direct style [121]. This type system still appears to be the most expressive one today [18, 233, 276, 364]. Constructing a list of the prefixes of a given list [97] provides one of the simplest examples illustrating the need for the full power of this type system [**41**, Section 3.6], and in fact so does the generator of Section 1.1.6 if one uses Hughes's higher-order representation of lists, as described in Section 2.2.2, page 41 [222]:

[6] For the longest time, the working title of "Essentials of Programming Languages" [182, 183] was "The Hitch-Hiker's Guide to the Meta-Universe".

```
(*  enumerate : 'a list -> ('a * 'a list) list  *)
fun enumerate xs
    = let (*  visit : 'a list * ('a list -> 'a list) -> ('a * 'a list) list  *)
          fun visit (nil, pc)
              = nil
            | visit (x :: xs, pc)
              = (x, pc xs) :: (visit (xs, fn ys => pc (x :: ys)))
      in visit (xs, fn ys => ys)
      end
```

Compared to Section 1.1.6, the type `'a list` has been replaced by the type `'a list -> 'a list`, the empty list has been replaced by the identity function, and the reversed prefix has been replaced by a function that prepends this prefix to its argument, thereby avoiding `reverse_prepend` altogether. In other words, the reversed prefix has been replaced by a prefix constructor.

One programmer's prefix constructor
is another programmer's delimited continuation.

The prefix constructor, i.e., the first parameter of `visit`, can be viewed as a delimited continuation and so `visit` could be written in direct style using shift and reset. The resulting program, however, would need the full power of the original type system for shift and reset [121] since the type of the intermediate answer is not the same as the type of the final answer in the definition of visit:

```
visit : ('a list -> 'a list) * 'a list -> ('a * 'a list) list
```

In any case, such a program can be written in Scheme, and it puts one in position to write a solution to the cryptarithmetic puzzle of Section 1.6 with an extra layer of continuations.

In his PhD thesis [170], Filinski shows that delimited continuations form a *universal* monadic computational effect. His ground-breaking implementation of shift and reset in terms of call/cc and one global cell (see page 21) is authoritative today [169].

The CPS hierarchy: Since his PhD thesis, Filinski extended his universality result to implementing hierarchies of layered monads in terms of the CPS hierarchy [171]. Zhe Yang and the author have also reported an operational investigation of the CPS hierarchy [150]. Małgorzata Biernacka, Dariusz Biernacki, and the author proposed an operational foundation for delimited continuations in the CPS hierarchy [**41**]. Kameyama has recently axiomatized layered delimited continuations in the CPS hierarchy [232, 233].

Static and dynamic delimited continuations: A number of operational alternatives to CPS have been sought to account for delimited continuations [159, 166, 167, 201, 215, 216, 271, 302, 328], starting with Felleisen's invention of control delimiters in direct style [163]. In these alternatives, a first-class delimited continuation is represented algebraically as a segment of the control stack. Delimiting control is achieved by marking the top of the control stack and abstracting control is achieved by dynamically traversing the stack in search for a particular mark (marks need not be unique) to construct a stack segment. Composing a captured continuation with the current continuation is achieved by concatenating the captured stack segment on top of the current control stack. Such an operational approach gives rise to a

number of choices, such as keeping or dropping stack marks when constructing and concatenating stack segments. These choices induce a variety of algebras of contexts that are still being explored today [186, 237, 323].

Viewed operationally, Felleisen's delimited-control operators `control` and `prompt` [163, 167], which only use one kind of stack mark, differ very little from `shift` and `reset`:

`control`: a `control`-abstracted stack segment is seamlessly concatenated to the current control stack;

`shift`: a mark is kept in between when concatenating a `shift`-abstracted stack segment to the current control stack.

However minor operationally, this difference has a major programming effect:

`control`: after seamless stack concatenation, the stack is dynamically accessible for control abstraction above and beyond the concatenation point;

`shift`: the stack is only statically accessible for control abstraction up to the concatenation point because a mark is kept at the concatenation point.

The following recursive descent over a list provides a non-trivial example where static and dynamic delimited continuations differ [**41**, Section 4.6]:

```
fun prepare_for_descent xs
    = let fun visit nil
              = nil
            | visit (x :: xs)
              = visit (call_with_current_delimited_continuation
                        (fn k => x :: (k xs)))
      in delimit_control (fn () => visit xs)
      end
```

where `call_with_current_delimited_continuation` stands for `shift` or `control` and `delimit_control` stands for `reset` or `prompt`. Using `shift` and `reset`, the input list is *copied*, whereas using `control` and `prompt`, the input list is *reversed*.

Moreover, the expressive power of dynamic delimited continuations makes it possible to write a *compositional* function traversing a binary tree in breadth-first order [49, **50**].

Simulating control operators in terms of one another: It is very simple to simulate `shift` and `reset` in terms of `control` and `prompt` [**47**, 122]: just delimit control at resumption points. For example, doing so in the example just above

```
fun prepare_for_descent' xs
    = let fun visit nil
              = nil
            | visit (x :: xs)
              = visit (call_with_current_delimited_continuation
                        (fn k => x :: (delimit_control (fn () => k xs))))
      in delimit_control (fn () => visit xs)
      end
```

yields a list-copying function, even when using `control` and `prompt`.

It, however, took 15 years for the converse simulation to see the light of day [323].[7] Since Ken Shan's simulation in 2004 [322], at least two more such simulations have been proposed, by Oleg Kiselyov [237] and by Biernacki, Millikin, and the author [**48**]. More generally, Kent Dybvig, Simon Peyton Jones, and Amr Sabry have proposed a basis of control operators in which to express other control operators [159].

Programming with delimited continuations: It is the author's consistent experience that CPS and the representation of contexts as 'defunctionalized' continuations (see Section 2.2 for more about defunctionalization) provide useful guidelines to design, reason about, and relate programs that use delimited continuations [**41**, Section 4.6] [**47**, Section 2.3]. For example, it is very simple to exhibit the CPS counterpart of the `shift/reset` version of `prepare_for_descent` [**48**] to figure it out in a purely functional way.

For this reason, Biernacki, Millikin, and the author have recently designed a dedicated 'dynamic CPS' that is compatible with defunctionalization [**48**]. This new form of CPS threads a trail of delimited continuations. Dynamic CPS makes it possible to account for dynamic delimited continuations with single marks (i.e., à la `control` and `prompt`) in principle, in theory, and in practice. For example, one can now exhibit the dynamic CPS counterpart of the `control/prompt` version of `prepare_for_descent` [**48**] to figure it out in a purely functional way. In addition, dynamic CPS is also compatible with Dybvig, Peyton Jones, and Sabry's more general architecture for dynamic delimited continuations with multiple marks [159], which is encouraging since this more general architecture was designed independently of defunctionalization.

[7] And then again, shortly before handing in this essay, an informal simulation of control and prompt in terms of shift and reset due to Filinski and dated 1994 emerged from the vault.

2 Program transformation

The transformations considered in this section concern block structure and lexical scope, the run-time representation of functions as values, and latent vs. manifest representations of control flow in the form of continuations.

2.1 Lambda-lifting and lambda-dropping

One function's parameter
is another function's free variable.

2.1.1 Background

Lambda-lifting was developed as part of compilers for lazy functional languages, in the 1980s [23, 221, 225], along with other transformations to handle lexically visible variables, e.g., supercombinator conversion [292]. Since then, it has been used in compilers [74, 303] and partial evaluators [57, 79, 253].

This program transformation caught the author's attention because it provides a systematic way to relate programs (such as the ones above) and thus a way out of considering them as merely different, even when one looks exotic. A tree should not hide the forest.

The fast exponentiation function for natural numbers, revisited. In Section 1.1.3, the local functions `visit` and `visit_positive` are defined in the scope of the first parameter of `power1`, `x`. This program is therefore *block-structured* and *scope-sensitive*: it is block-structured since it uses local functions, and it is scope-sensitive since `x` occurs free in the definitions of one of these local functions, `visit_positive`.

To make the program scope-insensitive, one lifts `x` from being a free variable in `visit_positive` to being a parameter of `visit_positive`, which forces `visit` to be passed `x` too:

```
fun power4 (x, n)                                  (* power4 : int * int -> int *)
    = let fun visit x n                             (* visit: int -> int -> int *)
              = if zero? n
                then 1
                else visit_positive x n
          and visit_positive x m       (* visit_positive : int -> int -> int *)
              = let val (q, r) = divmod2 m
                in if r
                   then sqr (visit_positive x q)
                   else mul (sqr (visit x q), x)
                end
      in visit x n
      end
```

To make the program non-block-structured, one lets the (now scope-insensitive) definitions of `visit` and `visit_positive` float to the same lexical level as `power4`:

```
fun power5_visit x n                      (* power5_visit : int -> int -> int *)
    = if zero? n
      then 1
      else power5_visit_positive x n
```

```
and power5_visit_positive x m   (* power5_visit_positive : int -> int -> int *)
    = let val (q, r) = divmod2 m
      in if r
         then sqr (power5_visit_positive x q)
         else mul (sqr (power5_visit x q), x)
      end

fun power5 (x, n)                                (* power5 : int * int -> int *)
    = power5_visit x n
```

The resulting program consists of three recursive equations.

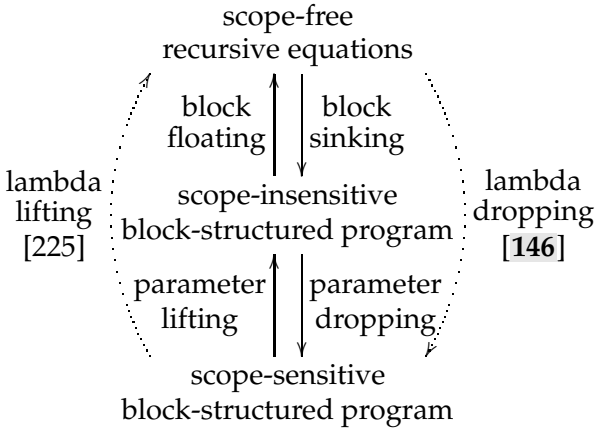
Conversely, one can let the definitions of `power5_visit` and of `power5_visit_positive` sink to their point of use in `power5`, obtaining the definition of `power4`, and one can drop `x` from being a parameter of these local functions to being a free variable in these local functions, obtaining the definition of `power1`.

Furthermore, one can let the definition of `visit_positive` sink to its point of use in `power1`:

```
fun power6 (x, n)                                (* power6 : int * int -> int *)
    = let fun visit n                                   (* visit : int -> int *)
              = let fun visit_positive m      (* visit_positive : int -> int *)
                        = let val (q, r) = divmod2 m
                          in if r
                             then sqr (visit_positive q)
                             else mul (sqr (visit q), x)
                          end
                in if zerop n
                   then 1
                   else visit_positive n
                end
      in visit n
      end
```

Indeed, the initial program was not maximally lambda-dropped, which occurs often in programming practice.

The overall situation is summarized as follows. Lifting free variables into function parameters and letting scope-insensitive lexical blocks float up into recursive equations realize lambda-lifting, and the converse transformations realize lambda-dropping:



2.1.2 Related work, contribution, impact, and future work

Lambda-lifting has been studied very little since its inception [225]. Ulrik Schultz and the author introduced its inverse transformation, lambda-dropping, and studied the impact of both transformations at compile time and at run time, their computational complexity, and their formal correctness. Schultz and the author also considered incremental versions of lambda-lifting and lambda-dropping because as illustrated in Section 2.1.1, programmers tend to write in a mixed style that is neither fully lambda-lifted nor fully lambda-dropped.

Overall, the investigation gave rise to Schultz's MS thesis [320], which was awarded a prize as the best 'Speciale' in Computer Science for 1997 in Denmark; to a conference publication and then a journal publication about lambda-dropping [145, **146**]; to a conference publication about the correctness of lambda-lifting and lambda-dropping [**109**]; and to a conference publication and then a journal publication about lowering the complexity of lambda-lifting from cubic time to quadratic time [147, **148**]. Furthermore, Peter Thiemann has studied the ML typeability of lambda-lifted programs [358], Adam Fischbach and John Hannan have formalized higher-order lambda-lifting and lambda-dropping in a logical framework [176], and Marco Morazán and Barbara Mucha are currently studying how to avoid extraneous parameters during lambda lifting while keeping its complexity quadratic [270].

Future work includes studying the logical foundations of lambda-lifting and lambda-dropping and their integration in a refactoring environment for block-structured programs.

2.2 Defunctionalization and refunctionalization

2.2.1 Background

In the early 1970's [304], John Reynolds introduced defunctionalization as a technical device that generalized Peter Landin's representation of closure [243], where a term is paired together with its environment. In a defunctionalized program, what is paired with an environment is not a term, but a tag that determines this term uniquely. Reynolds then never used defunctionalization again [306], save for deriving a first-order semantics in his textbook on programming languages, in the late 1990's [307, Section 12.4].

Example: the sine function. Let us go back to the version of the sine function on page 11:

```
fun sin1 (x, epsilon)
    = let fun visit_c (x, k)     (* visit_c : real * (real -> real) -> real *)
              = if abs x < epsilon
                then k x
                else visit_c (x / 3.0,
                              fn s => k (3.0 * s - 4.0 * s * s * s))
      in visit_c (x, fn s => s)
      end
```

The continuation of `visit_c` is an inhabitant of the function space "`real -> real`". More precisely, it is an instance of one of the following two function abstractions:

- `fn s => s`, which is closed; and
- `fn s => k (3.0 * s - 4.0 * s * s * s)`, which has one free variable, `k`.

To implement the continuation, one therefore does not need the full function space "`real -> real`": a sum with two first-order summands is enough, provided that each summand can be interpreted properly. One can therefore manufacture a data type `cont` with two constructors together with an apply function. Each constructor represents one of these function abstractions, together with the values of its free variables, and the apply function dispatches on which constructor is being applied:

```
datatype cont = C0
              | C1 of cont

(*  apply_cont : cont * real -> real  *)
fun apply_cont (C0, s)
    = s
  | apply_cont (C1 k, s)
    = apply_cont (k, 3.0 * s - 4.0 * s * s * s)
```

The defunctionalized version of the sine function then reads as follows. Each function introduction (i.e., declaration) is replaced by an injection into the data type `cont`, and each function elimination (i.e., application) is replaced by a call to `apply_cont`:

```
fun sin2 (x, epsilon)
    = let fun visit (x, k)                    (* visit : real * cont -> real *)
              = if abs x < epsilon
                then apply_cont (k, x)
                else visit (x / 3.0, C1 k)
      in visit (x, C0)
      end
```

Most of the time, the defunctionalized version of a function and the apply function are mutually recursive.

2.2.2 Related work, contribution, impact, and future work

Although the term "defunctionalization" never took root,
the ideas behind it have become commonplace.
John Reynolds, 1998 [306]

In the early 1980's, i.e., ten years after "Definitional Interpreters" [304], David Warren independently proposed defunctionalization as a simple device to make logic programming higher-order [376]. Or rather: he wondered about the need for higher-order extensions of Prolog since this transformation provides a simple encoding of higher-order features.

Defunctionalization is used considerably less than, e.g., closure conversion [14, 17, 240, 243, 338]: Anders Bondorf uses it to make higher-order programs amenable to first-order partial evaluation [54]; Andrew Tolmach and Dino Oliva use it to translate ML programs into Ada [361]; Leonidas Fegaras uses it in his object-oriented database management system, lambda-DB [160]; Daniel Wang and Andrew Appel use it in type-safe garbage collectors [375]; and it is used in Urban Boquist's Haskell compiler [60], MLton [67], and Hilog [71] to represent functions as values. Recently though, defunctionalization has undergone a renewal of interest, as Shriram Krishnamurthi, Matthias Felleisen, and their colleagues and students use it for web programming [193, 192, 290].

Only lately has defunctionalization been formalized: Jeffrey Bell, Françoise Bellegarde, and James Hook showed that it preserves types [33]; Lasse Nielsen proved its partial correctness using denotational semantics [279, 280]; Anindya Banerjee, Nevin Heintze and Jon Riecke proved its total correctness using operational semantics [29]; and recently, François Pottier and Nadji Gauthier have shown that polymorphically typed programs could be defunctionalized into first-order ones that use guarded algebraic data types [300].

In the early 2000's, Nielsen and the author set out to show that defunctionalization is not only useful as an implementation technique and interesting as a theoretical topic of study: it can also play a role when programming, when reasoning about programs, when analyzing programs, when transforming programs, and when specifying programming languages [**136**]. Each of these points is developed in the rest of this section.

Defunctionalization for programming: To illustrate that defunctionalization connects independently known programs of independent interest, let us consider John Hughes's higher-order representation of lists [222], where the empty list is represented with the identity function and the list constructor is curried:

```
(*  ho_nil :        'a list -> 'a list  *)
val ho_nil
    = fn ys => ys

(*  ho_cons : 'a -> 'a list -> 'a list *)
fun ho_cons y
    = fn ys => y :: ys
```

In effect, Hughes uses the monoid of list transformers rather than the monoid of lists. The key advantage of one over the other is that in the monoid of lists, composition is implemented by list concatenation, i.e., in linear time over its first argument, whereas in the monoid of list transformers, composition is implemented by function composition, i.e., in constant time. Hughes showed that the naive, concatenation-based version of the reverse function in the monoid of list transformers is as efficient as the concatenation-less, accumulator-based one in the monoid of lists. Nielsen and the author confirmed the effectiveness of the representation by showing that a defunctionalized version of the naive, concatenation-based version of the reverse function in the monoid of list transformers corresponds to the concatenation-less, accumulator-based one in the monoid of lists [**136**, Section 2.2].

To illustrate this correspondence, let us write the generator of Section 1.4.3, page 8 with a curried list transformer:

```
fun generate_higher_order (xs, lazy_nil, lazy_cons)
    = let (*  visit : 'a list * ('a list -> 'a list) -> 'b  *)
          fun visit (nil, pc)
              = lazy_nil ()
            | visit (x :: xs, pc)
              = lazy_cons ((x, pc xs),
                           fn () => visit (xs, fn a => pc (x :: a)))
      in visit (xs, fn a => a)
      end
```

Instead of being parameterized by a reversed list prefix, `visit` is parameterized by a function that, applied to `nil`, constructs this list prefix, and that, applied to another list, prepends this prefix to this other list.

Let us defunctionalize the higher-order representation of lists in the definition of `generate_higher_order`. The function space `'a list -> 'a list` is inhabited by instances of the two shaded function abstractions. The corresponding sum space and its apply function therefore read as follows:

```
datatype 'a prefix_constructor = PC0
                               | PC1 of 'a * 'a prefix_constructor

fun apply_prefix_constructor (PC0, a)
    = a
  | apply_prefix_constructor (PC1 (x, pc), a)
    = apply_prefix_constructor (pc, x :: a)
```

In the following defunctionalized version of `generate`, the injections and projection of the sum space are shaded:

```
fun generate_higher_order_defunct (xs, lazy_nil, lazy_cons)
    = let (*  visit : 'a list * prefix_constructor -> 'b  *)
          fun visit (nil, pc)
              = lazy_nil ()
            | visit (x :: xs, pc)
              = lazy_cons ((x, apply_prefix_constructor (pc, xs)),
                           fn () => visit (xs, PC1 (x, pc)))
      in visit (xs, PC0)
      end
```

One can observe that the data type `prefix_constructor` is isomorphic to that of lists and that the dispatch function `apply_prefix_constructor` is equivalent to `reverse_prepend` on page 8, modulo this isomorphism. The two versions `generate` page 8 and `generate_higher_order` above therefore do correspond to each other in that the former is a defunctionalized version of the latter.[8] In a similar vein, defunctionalizing an interpreter for regular-expressions [122, 203] yields a pushdown automaton with two stacks [**136**, Section 5]. Section 2.4 contains more examples.

Defunctionalization for reasoning about programs: Often, defunctionalization connects a higher-order program that is compositional in its input and a first-order program that is not. Nielsen and the author have compared the correctness proofs of two interpreters for regular expressions that correspond to each other by defunctionalization [**136**, Section 5]. One proof proceeds by structural induction (using an auxiliary relation) and the other proceeds by well-founded induction. It should be possible to use defunctionalization for deriving the ordering—a future work.

Defunctionalization for analyzing programs: Defunctionalization is a natural consumer of control-flow information as provided by 0CFA [325], namely: which λ-abstraction in a program gives rise to a closure that may flow to a given application site, and to which application sites may a closure arising from a given λ-abstraction flow to [123, Section 2]? This information determines how to partition a function space into a sum and how to write the corresponding apply function. After control-flow analysis (and defunctionalization), a program can be subjected to further analyses, e.g., binding-time analysis [54, 283].

[8] Hindsight is such an exact science.

Defunctionalization for transforming programs: In his classical work on continuation-based program transformation strategies [367], Mitchell Wand presents a method in which

1. a given program is transformed into continuation-passing style (see Section 2.3);
2. a data-structure is designed to represent the continuation; and
3. this representation is used to improve the initial program.

In each of the examples mentioned in Wand's article, the designed data structure can be obtained by defunctionalizing the continuation. This coincidence is significant because finding such "data-structure continuations" is listed as the main motivation for the method [367, page 179]. Of course, there probably are many other ways to design data-structure continuations, but the point here is that defunctionalization fits the bill.

Wand's work is seminal in that it has shown that in effect, defunctionalizing a CPS-transformed first-order program provides a systematic way to construct an iterative version of this program that uses a push-down accumulator. The representation of the accumulator can then be changed to improve the efficiency of the original program. For example, let us get back to the data type of defunctionalized continuations on page 40:

```
datatype cont = C0
              | C1 of cont
```

It is isomorphic to Peano numbers. One can thus represent a defunctionalized continuation as a native ML integer:

```
type cont = int
val C0 = 0
val C1 = fn k => k + 1

fun apply_cont (0, s)
    = s
  | apply_cont (k, s)
    = apply_cont (k - 1, 3.0 * s - 4.0 * s * s * s)

fun sin3 (x, epsilon)
    = let fun visit (x, k)
              = if abs x < epsilon
                then apply_cont (k, x)
                else visit (x / 3.0, C1 k)
      in visit (x, C0)
      end
```

The result is an automaton with a counter: `sin3` iteratively increments the counter while dividing the first parameter by 3; and then `apply_cont` iteratively decrements the counter while incrementally computing the result. In other words, `sin3` computes the smallest k such that $\frac{x}{3^k} < \varepsilon$, i.e., such that $x < 3^k \times \varepsilon$, and it does so in $O(k)$ time and $O(1)$ space, leaving room for improvement—which illustrates the benefit of relying on continuations for transforming programs, as advocated by Wand [367].

Defunctionalization for specifying programming languages: The notion of context (term with a hole) is a standard one in the λ-calculus [30, page 29]. The notions of reduction contexts and evaluation contexts have been put forward by Matthias Felleisen in his PhD thesis [161] as foundational ones to specify calculi and programming languages with effects, and these notions are accepted as such today [294]. It is however surprisingly tricky to write grammars of reduction and evaluation contexts, and then to establish, e.g., a unique-decomposition property. That said,

- given a compositional function that recursively descends into a term in search for the next redex according to a reduction strategy, one can transform it into continuation-passing style and defunctionalize its continuation: as observed by Nielsen and the author, the resulting data type is the grammar of reduction contexts, and the unique-decomposition property follows from compositionality;
- given a compositional evaluation function for terms, one can transform it into continuation-passing style and defunctionalize its continuation: as also observed by Nielsen and the author, the resulting data type is the grammar of evaluation contexts;
- furthermore, the two grammars (of reduction contexts and of evaluation contexts) are identical.

So to a considerable extent, continuations and contexts come hand in hand, not only extensionally but intensionally and constructively, using defunctionalization [**114**].

As reported in the rest of this essay, the author and his students have made an abundant use of this triple observation for connecting calculi, abstract machines, and evaluation functions.

2.3 CPS and direct-style transformations

2.3.1 Background

This section addresses the transformation of functional programs into continuation-passing style (CPS). In the author's eyes, the CPS transformation is an idealized program transformation in the same sense that the lambda-calculus is an idealized programming language: any progress in understanding the CPS transformation (and the lambda-calculus) is a progress in understanding program transformations (and programming languages).

The CPS transformation has been variously discovered and used [305]. For example, van Wijngaarden presented it as a device to eliminate jumps in Algol programs [362]; Fischer used it to demonstrate that a deletion strategy incurs no loss of expressive power compared to a retention strategy for representing a control stack [177]; and Steele used a two-pass CPS transformation in the first Scheme compiler [338].

A CPS transformation encodes an evaluation order [206]. As most lambda-encodings, CPS transformations generate many administrative redexes. Optimizing these administrative redexes is done in one or two passes, i.e., during the CPS transformation or immediately afterwards.

2.3.2 Related work, contribution, impact, and future work

CPS transformation of terms: Andrzej Filinski and the author presented a one-pass CPS transformation that is higher order and properly tail recursive [122, 123]. Being properly tail recursive means that the continuation of a CPS-transformed tail call is k, not $\lambda v.kv$, for some k. The CPS transformation of Standard ML of New Jersey now integrates this optimization. John Hatcliff and the author presented one-pass CPS transformations that exploit the results of strictness analysis [126] and that exploit the results of totality analysis [127]. Hatcliff and the author also factored the call-by-name CPS transformation into a thunk introduction (to simulate call by name in call by value) and a call-by-value CPS transformation extended to handling thunks [208]. Kristian Støvring recently used this thunk introduction to study higher-order matching [344]. Lasse Nielsen and the author presented a one-pass CPS transformation that integrates generalized beta reduction [140]. Observing that existing CPS transformations only satisfy two out of the three properties of being first-order, one-pass, and compositional, Nielsen and the author presented a CPS transformation that satisfies all three properties [**138**]. The author extended the one-pass CPS transformation to short-cut boolean expressions [**112**]. Filinski developed an extensional CPS transformation operating on values instead of on terms [173].

Direct-style transformation: The author presented a left inverse to the CPS transformation, the direct-style transformation [**100**]. Julia Lawall and the author extended the direct-style transformation to first-class continuations [130] and staged the CPS transformation and the direct-style transformation using the guidelines of Galois connections [249], an approach that Amr Sabry and Philip Wadler successfully lifted to a semantic level [314]. John Hatcliff and the author factored CPS transformations and direct-style transformations through Eugenio Moggi's computational metalanguage [206].

CPS transformation of types: Mitchell Wand and Albert Meyer showed that the CPS transformation over terms induces a CPS transformation over types [259, 369]. Tim Griffin identified that through the Curry-Howard isomorphism, the CPS transformation corresponds to a double-negation translation [196], and Chet Murthy identified how known double-negation translations correspond to known CPS transformations encoding one evaluation order or another [275]. Section 5.2.11 mentions another application [99].

CPS transformation of flow information: The CPS transformation over terms also induces a CPS transformation over flow information. Jens Palsberg and Mitchell Wand described this analogue for control-flow analysis [288], using the Socratic questions and answers of "Representing Control" [123, Section 2] as an explanatory device. Daniel Damian and the author complemented Palsberg and Wand's CPS transformation of flow information with the analogue of administrative reductions as well as with the corresponding one-pass transformation [**94**]. Damian and the author also presented two CPS transformations of binding-time information [**95**] and a simple CPS transformation of control-flow information based on Nielsen and the author's first-order, one-pass, and compositional CPS transformation mentioned above [93]. Section 4.2.2 elaborates on the usefulness of CPS-transforming flow information.

CPS transformation for what? Very early, Guy Steele identified that CPS-transforming a source program brought to the syntactic surface much of what a compiler needs to do anyway, namely naming intermediate results and sequentializing their computation [337, 338, 339]. As pointed out by Gordon Plotkin [295], a CPS-transformed program can be reasoned about using simply beta-reduction. Amr Sabry and Matthias Felleisen presented the corresponding axioms (including contextual ones) in direct style [312], using an untyped version of Eugenio Moggi's computational metalanguage [268]. Hatcliff and the author pointed out that administrative reductions in CPS terms correspond to monadic reductions in the computational metalanguage [207], and that CPS terms, after administrative reductions, are in one-to-one correspondence with monadic normal forms (sometimes called A-normal forms). Monadic normal forms have been proposed as an alternative to CPS terms in compilers [179, 241, 303, 325, 326].

Independently of compiling, the CPS transformation can also be used as a prelude to further transformations, as illustrated next.

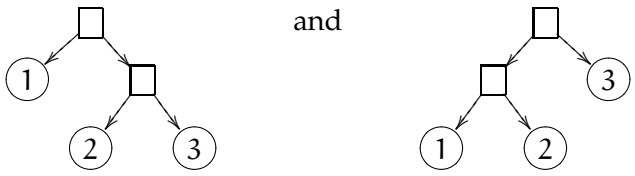
2.4 Synergy

The journal version of "There and Back Again" [**124**] uses the CPS transformation, defunctionalization, and their left inverses to provide entry points for writing programs that recursively traverse one data structure at call time and iteratively traverse another one at return time. The CPS transformation yields programs that inductively build, at call time, a continuation that iteratively traverses the other data structure. Defunctionalization provides a first-order representation of this continuation as a data structure (an accumulator).

The rest of this section briefly reviews other situations where CPS transformation and defunctionalization synergize, and concludes with two alternative fast exponentiation functions for natural numbers.

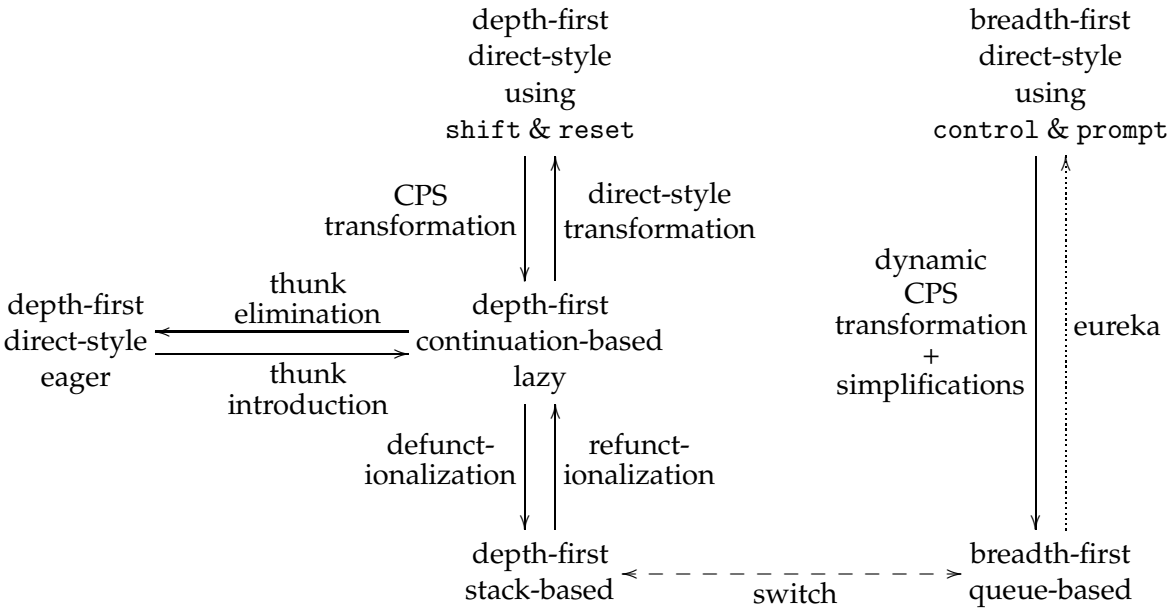
2.4.1 The samefringe problem

The samefringe problem is traditionally stated as follows: do two given trees of integers have the same sequence of leaves when read from left to right? For example, the two trees



have the same fringe `[1,2,3]` (representing it as a list) even though they are shaped differently. Because of this difference of shape, each tree needs to be traversed independently of the other. Furthermore, because the computation can stop as soon as two leaves differ in the fringes, a more efficient solution is sought than the naive one that (1) recursively flattens each tree into a list, and (2) iteratively traverses the two resulting lists. Among the two best-known solutions, one flattens each tree into a lazy list [211] and the other flattens each tree incrementally [255]. Many other solutions exist, involving generators, threads, coroutines, control operators, etc.

In their recent work on static and dynamic delimited continuations [**50**], Biernacki, Shan, and the author (1) take a transformational approach to the samefringe problem and (2) introduce its breadth-first counterpart:



- The starting point (left side of the diagram) is the naive, eager, and compositional program in direct style flattening a tree into a list. This program can be made lazy by using thunks, i.e., functions of type "`unit -> 'a`" (center of the diagram).
- Going up in the diagram, the type "`unit -> 'a`" can be viewed not as a functional device to implement laziness but as a unit-accepting delimited continuation.

  One programmer's suspension
  is another programmer's delimited continuation.

  The lazy compositional program is then, in actuality, a continuation-based one, and one that is the CPS counterpart of a lazy compositional program in direct style that uses the static delimited-control operators `shift` and `reset` (top center of the diagram).
- Going down in the diagram, defunctionalizing the thunks in the lazy compositional program yields a non-compositional program (bottom center of the diagram). This program uses a stack of subtrees and coincides with McCarthy's solution [255].
- The stack-based non-compositional program (bottom center of the diagram) implements a depth-first traversal. Replacing the stack with a queue yields a queue-based non-compositional program implementing a breadth-first traversal (bottom right of the diagram).
- By analogy with the rest of the diagram, Biernacki, Shan, and the author inferred a compositional program in direct style that uses the dynamic delimited-control operators `control` and `prompt` (top right of the diagram) and that corresponds to the queue-based non-compositional program.

  Since then, Biernacki, Millikin, and the author presented a 'dynamic CPS transformation' giving rise to the bottom-right program given the top-right one [**48**].

2.4.2 Lightweight defunctionalization

A 'deep closure' pairs a term and a complete environment, whereas a 'flat closure' pairs a term and the values of its free variables [16, 17, 63, 241, 243, 254]. Sometimes, however, there is no need to put the value of a free variable in a flat closure if this closure is always applied in the scope of this variable. This observation motivated Paul Steckler and Mitchell Wand's study of *lightweight closure conversion* [336]. A similar situation holds for defunctionalization.

To illustrate lightweight defunctionalization, let us go back to the fast exponentiation function and consider a version with a direct-style interface and two local functions in CPS:

```
fun power7 (x, n)                                  (* power7 : int * int -> int *)
    = let fun visit_c (n, k)           (* visit_c : int * (int -> int) -> int *)
              = if zero? n
                then k 1
                else visit_positive_c (n, k)
                                (* visit_positive : int * (int -> int) -> int *)
          and visit_positive_c (m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive_c (q, fn v => k (sqr v))
                   else visit_c (q, fn v => k (mul (sqr v, x)))
                end
      in visit_c (n, fn v => v)
      end
```

The continuation is an inhabitant of the function space "`int -> int`". More precisely, it is an instance of one of the following three function abstractions:

- `fn v => v`, which is closed;
- `fn v => k (sqr v)`, which has one free variable, `k`; and
- `fn v => k (mul (sqr v, x))`, which has two free variables, `k` and `x`.

To represent the continuation in a first-order fashion, one can therefore manufacture a data type `cont` with three constructors together with an apply function. Each constructor represents one of these function abstractions, together with the values of its free variables, and the apply function dispatches on which constructor is being applied:

```
datatype cont = C0
              | C1 of cont
              | C2 of cont * int

fun apply_cont (C0, v)
    = v
  | apply_cont (C1 k, v)
    = apply_cont (k, sqr v)
  | apply_cont (C2 (k, x), v)
    = apply_cont (k, mul (sqr v, x))
```

The defunctionalized version of the power function then reads as follows. Each function declaration is replaced by an injection into the data type `cont`, and each function application is replaced by a call to `apply_cont`:

```
fun power8 (x, n)
    = let (*  visit_c : int * cont -> int  *)
          fun visit_c (n, k)
              = if zerop n
                then apply_cont (k, 1)
                else visit_positive_c (n, k)
          (*  visit_positive_c : int * cont -> int  *)
          and visit_positive_c (m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive_c (q, C1 k)
                   else visit_c (q, C2 (k, x))
                end
      in visit_c (n, C0)
      end
```

This power function has been defunctionalized on the grounds that the function abstraction "`fn v => k (mul (x, sqr v))`" has two free local variables, `k` and `x`. However, the definition of `power7` is in lambda-dropped form, and so `x` *is in the scope of both the introduction and the elimination of all the continuations*. There is therefore no need for an integer placeholder in the first-order representation of the continuations if the corresponding apply function is defined in the scope of `x`:

```
datatype cont = C0
              | C1 of cont
              | C2 of cont  (* no int *)

fun power9 (x, n)
    = let fun apply_cont (C0, v)
              = v
            | apply_cont (C1 k, v)
              = apply_cont (k, sqr v)
            | apply_cont (C2 k, v)
              = apply_cont (k, mul (sqr v, x))
          fun visit_c (n, k)
              = if zerop n
                then apply_cont (k, 1)
                else visit_positive_c (n, k)
          and visit_positive_c (m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive_c (q, C1 k)
                   else visit_c (q, C2 k)
                end
      in visit_c (n, C0)
      end
```

This lightweight representation of the 'data continuation' is isomorphic to strictly positive natural numbers. One can therefore switch to using native ML integers:

```
type cont = int
val C0 = 1
val C1 = fn k => 2 * k
val C2 = fn k => 2 * k + 1
```

The continuation is now constructed by multiplication and deconstructed by division, using the remainder to distinguish which of `C1` or `C2` was used for the corresponding construction:

```
fun power10 (x, n)
    = let fun apply_cont (1, v)
              = v
          | apply_cont (k, v)
            = let val (k', r) = divmod2 k
              in if r
                 then apply_cont (k', sqr v)
                 else apply_cont (k', mul (sqr v, x))
              end
          fun visit_c (n, k)
              = if zero? n
                then apply_cont (k, 1)
                else visit_positive_c (n, k)
          and visit_positive_c (m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive_c (q, C1 k)
                   else visit_c (q, C2 k)
                end
      in visit_c (n, C0)
      end
```

The resulting fast exponentiation function operates in two iterations:

1. in the first iteration, `visit_c` and `visit_positive_c` reverse the binary representation of the exponent and prepend 1 to the result; for example, they map 1110_2 (i.e., 14 in base 10) to 10111_2 (i.e., 23 in base 10);
2. in the second iteration, `apply_cont` traverses the reversed representation.

In HAKMEM [32], Item 167 contains a discussion about how to reverse the binary representation of a number efficiently. This possibility again illustrates the benefit of relying on continuations for transforming programs [367], as on page 43.

2.4.3 Two alternative fast exponentiation functions for natural numbers

Re-associating the multiplications. The following alternative identities also match the BNF of Section 1.1.3, page 5:

$$\begin{aligned} x^0 &= 1 \\ x^{2m} &= (x^2)^m \\ x^{2n+1} &= (x^2)^n \times x \end{aligned}$$

In the corresponding alternative definition, all parameters take an active part in the computation and therefore none can be dropped, unlike in Section 2.1:

```
fun power1_alt (x, n)
    = let fun visit (y, n)
              = if zero? n
                then 1
                else visit_positive (y, n)
```

```
          and visit_positive (y, m)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive (sqr y, q)
                   else mul (visit (sqr y, q), y)
                end
      in visit (x, n)
      end
```

CPS-transformation and defunctionalization yield a program where the defunctionalized continuations are isomorphic to lists of integers. One can thus represent a defunctionalized continuation as a native ML list:

```
type cont = int list
val C0 = nil
val C1 = fn (k, y) => y :: k

(*  apply_cont : cont * int -> int  *)
fun apply_cont (nil, v)
    = v
  | apply_cont (y :: k, v)
    = apply_cont (k, mul (v, y))

fun power1_alt' (x, n)
    = let (*  visit : int * int * cont -> int  *)
          fun visit (y, n, k)
              = if zerop n
                then apply_cont (k, 1)
                else visit_positive (y, n, k)
          and visit_positive (y, m, k)
              = let val (q, r) = divmod2 m
                in if r
                   then visit_positive (sqr y, q, k)
                   else visit (sqr y, q, C1 (k, y))
                end
      in visit (x, n, C0)
      end
```

The resulting fast exponentiation function operates in two iterations. The first iteration, implemented by `visit` and `visit_positive`, accumulates a list continuation which is as long as the number of odd bits in the binary representation of the given exponent. The second iteration, which is implemented by `apply_cont`, folds this list into as many multiplications.

Introducing an accumulator. Finally, the non-tail recursive calls to `visit`, in `power1_alt`, generate a trail of multiplications, just as in the traditional factorial function. Since multiplication is associative, an accumulator can be used instead:

```
fun power2_alt (x, n)
    = let (*  visit : int * int * int -> int  *)
          fun visit (y, n, a)
              = if zerop n
                then a
                else visit_positive (y, n, a)
```

```
    and visit_positive (y, m, a)
        = let val (q, r) = divmod2 m
          in if r
             then visit_positive (sqr y, q, a)
             else visit (sqr y, q, mul (y, a))
          end
in visit (x, n, 1)
end
```

The resulting fast exponentiation function operates in one iteration and corresponds to the following identities:

$$\begin{aligned} x^0 \times a &= a \\ x^{2m} \times a &= (x^2)^m \times a \\ x^{2n+1} \times a &= (x^2)^n \times x \times a \end{aligned}$$

Since it is iterative, there is no point to CPS-transform this version.

Inlining one of the local functions. In each of the versions of the fast exponentiation function, either `visit` or `visit_positive` can be inlined (see Section 5.2.10, page 85 for an example). Each of the inlined functions operates in as many recursive calls to the remaining local function as the number of bits in the binary representation of the given exponent.

3 Program execution

This section addresses and connects three approaches to executing programs. A program is a λ-term, usually with literals, and executing it consists in computing its weak head normal form.

3.1 Background

The three approaches to executing programs are iterating one-step reduction, using an abstract machine, and using an evaluation function.

3.1.1 One-step reduction

As developed in the accompanying articles [**41**, **42**, **43**, **139**], one-step reduction is specified based on

- a BNF of terms,
- a collection of reduction rules specifying how to contract redexes, and
- a collection of compatibility rules specifying how to deterministically navigate in the term in search of the next redex.

A one-step reduction function maps a non-value term to a term where the next redex in the reduction sequence has been contracted.

As an alternative to this classical specification [30, 295], Matthias Felleisen proposed to replace the compatibility rules with a grammar of reduction contexts [161, 164]. Rather than recursively constructing a proof tree from a given term towards a redex, using the compatibility rules, a reduction context is accumulated iteratively from term to redex, guided by the compatibility rules [260].

In both cases, evaluation is defined as the reflexive and transitive closure of one-step reduction.

3.1.2 Abstract machines

As developed in the accompanying articles [**5**, **6**, **7**, **41**, **42**, **43**, **46**, **47**, **48**, **50**, **115**, **135**, **139**], an abstract machine is specified based on

- a BNF of terms,
- state-transition functions implementing the reduction rules and the compatibility rules, based on a grammar of evaluation contexts.

These machines take their formal roots in operational semantics [155, 164, 296]

3.1.3 Evaluation functions

As developed in the accompanying articles [**5**, **6**, **7**, **41**, **41**, **46**, **48**, **115**, **135**], an evaluation function is specified based on

- a BNF of terms,
- a compositional function mapping a term to a value.

These evaluation functions take their formal roots in denotational semantics [317, 342, 304].

3.2 Contribution, impact, and future work

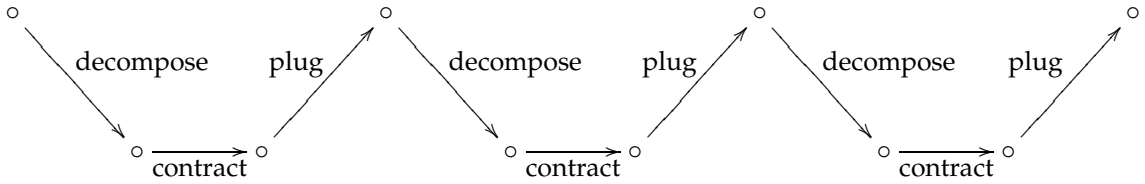
This section builds on the observation mentioned in Section 2.2.2, page 44 that reduction contexts and evaluation contexts coincide, and that together with a suitable apply function, they are defunctionalized continuations for a one-step reduction function and for an evaluation function [**114**]. Section 3.2.1 exploits the connection between reduction contexts and continuations with a syntactic correspondence between one-step reduction and abstract machines. Section 3.2.2 exploits the connection between evaluation contexts and continuations with a functional correspondence between evaluation functions and abstract machines. Section 3.2.3 illustrates how the syntactic correspondence and the functional correspondence synergize. The functional correspondence amplifies John Reynolds's seminal work reported in "Definitional Interpreters for Higher-Order Programming Languages" [304], and Section 3.2.4 presents a corollary of the evaluation-order dependence pointed out by Reynolds.

3.2.1 Connecting one-step reduction and abstract machines: a syntactic correspondence

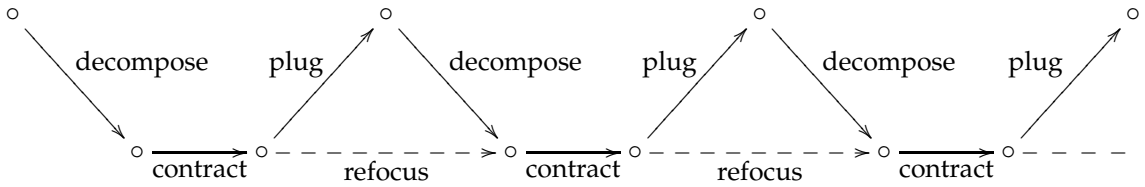
The syntactic correspondence is motivated by the inefficiency of implementing evaluation as the reflexive-transitive closure of one-step reduction [321, 378, 379]. Indeed, in a reduction semantics, a non-value term is reduced by

1. decomposing it into a redex and its context,
2. contracting the redex, and
3. plugging the contractum into the reduction context, which yields a new term.

An evaluation function is defined using the reflexive and transitive closure of the one-step reduction function:



Nielsen and the author made the simple observation that in the composition of plug and decompose, the intermediate terms could be avoided by fusing the composition into a 'refocus' function going from redex site to redex site [137, **139**]:



The resulting 'refocused' evaluation function is defined as the reflexive and transitive closure of refocusing and contraction.

What started as a mere program optimization turned out to have a number of applications. Nielsen and the author stated general conditions over the structure of a reduction semantics for a refocus function to be mechanically constructable, and they presented an algorithm to construct such a refocus function [**139**]. These general conditions are mild (essentially the reduction semantics should satisfy a unique decomposition property); in fact, they already are assumed in Xiao et al.'s implementation work [378, 379], which motivated the present study.

Refocusing has been applied to a variety of situations:

- to context-based CPS transformations [312], improving them from operating in quadratic time to operating in one pass [**139**];
- to specifications of the λ-calculus with traditional substitutions [**139**], mechanically obtaining a variety of known and new abstract machines, including the CK machine [164];
- to specifications of the λ-calculus with explicit substitutions [**43**], mechanically obtaining a variety of known and new abstract machines with environments, including the CEK machine [165], the Krivine machine [90], the original version of Krivine's machine [242], and the Zinc machine [251].
- to specifications of the λ-calculus with explicit substitutions and computational effects [**42**], mechanically obtaining a variety of known and new abstract machines from known calculi and a variety of new calculi from known abstract machines for Krivine's machine with call/cc, the λμ-calculus, delimited continuations, i/o, stack inspection, proper tail-recursion, and lazy evaluation.

3.2.2 Connecting evaluation functions and abstract machines: a functional correspondence

One programmer's abstract machine
is another programmer's defunctionalized interpreter.

The functional correspondence takes its roots in the realization that often, an abstract machine is the defunctionalized counterpart of a CPS program [**115**] and that defunctionalizing a CPS program yields an abstract machine [**5**, **6**, **46**].

Often, an abstract machine is the defunctionalized counterpart of a CPS program: What started this line of work was yet another attempt to understand the venerable SECD machine [243].[9] For a programmer, one of the reasons for the conceptual complexity of the SECD is that its transition function has several 'induction variables', i.e., it is defined by cases over several of its four parameters—a data stack containing intermediate values (of type `S`), an environment (of type `E`), a control stack (of type `C`), and a dump (of type `D`):

```
run : S * E * C * D -> value
```

This transition function can be disentangled into four (mutually recursive) ones, each being defined by cases over one of its parameters:

[9] *In mathematics one does not understand things. One just gets used to them.*
John von Neumann

```
run_c :        S * E * C * D -> value
run_d :                S * D -> value
run_t : term * S * E * C * D -> value
run_a :        S * E * C * D -> value
```

This disentanglement opens the door to a series of technical miracles [**115**]:[10]

1. the disentangled definition is in defunctionalized form: the control stack and the dump are defunctionalized data types, and `run_c` and `run_d` are the corresponding apply functions; it thus can be refunctionalized, thereby eliminating the two apply functions:

```
run_t : term * S * E * C * D -> value
run_a :        S * E * C * D -> value
where  C = S * E * D -> value  and  D = S -> value
```

2. the refunctionalized definition is in CPS, with two layered continuations; it thus can be mapped back to direct style, eliminating the dump continuation:

```
run_t : term * S * E * C -> S
run_a :        S * E * C -> S
where C = S * E -> S
```

3. the direct-style definition is in continuation-composing style; it thus can be mapped back to direct style with one control delimiter (a moot one, since there is no corresponding control abstraction), eliminating the control continuation:

```
run_t : term * S * E -> S * E
run_a :        S * E -> S * E
```

4. due to the closures, the direct style definition is again in defunctionalized form, and it thus can be refunctionalized, changing the data type of values from using closures

```
datatype value = INT of int | CLO of (identifier * term) * E
```

to using functions: `datatype value = INT of int | FUN of value -> value`

The result is an evaluation function that is (1) compositional, (2) in direct style, (3) with a moot control delimiter, (4) managing its environment in a callee-save fashion, and (5) threading a data stack of intermediate results. It is simple to eliminate the explicit data stack:

```
run_t :          term * E -> value * E
run_a : value * value * E -> value * E
```

The result is a compositional evaluation function in direct style, with a control delimiter, and managing its environment in a callee-save fashion. Since the control delimiter is moot, it can be dispensed with. The resulting evaluation function is the callee-save counterpart of Lockwood Morris's (caller-save) evaluation function [273] as considered by Gordon Plotkin for his original correctness proof of the SECD machine [295]:

```
 eval :          term * E -> value
apply : value * value * E -> value
```

[10] Even keeping in mind that to a man with a hammer, the world looks like a nail, these technical miracles may remind one of Godfrey Hardy's description of a mathematical proof: "there is a very high degree of *unexpectedness*, combined with *economy* and *inevitability*. The arguments take so odd and surprising a form; the weapons used seem so childishly simple when compared with the far-reaching results; but there is no escape from the conclusion. There are no complications of detail—one line of attack is enough in each case;" [202, § 18].

All the steps above are reversible (making it possible to obtain the original SECD machine),[11] and there is plenty of room for variation (making it possible to obtain SECD-machine variants). More significantly though, considering these steps in reverse—i.e., defunctionalization of function values into closures, CPS transformation, and defunctionalization of continuations—should be familiar since most of them occur in Reynolds's original article [304], in that order. The apple does not fall far from the tree.

Defunctionalizing a CPS program yields an abstract machine: Because CPS programs are tail-recursive, all calls are tail calls [338]; and because CPS programs are evaluation-order independent, all sub-computations are trivial [295, 304]. Therefore defunctionalizing a CPS program also yields a (first-order) program where all calls are tail calls and all sub-computations are trivial—in other words, a program with state-transition functions, i.e., an abstract machine.

Now the interesting question—one that the author tackled together with Mads Sig Ager, Małgorzata Biernacka, Dariusz Biernacki, and Jan Midtgaard—is: which evaluation functions yield which abstract machines?

- CPS-transforming and defunctionalizing the traditional call-by-value evaluation function for the λ-calculus yields the CEK machine [**5**].
- CPS-transforming and defunctionalizing the traditional call-by-name evaluation function for the λ-calculus (which uses thunks) yields the Krivine machine [**5**].
- CPS-transforming and defunctionalizing the traditional call-by-need evaluation function for the λ-calculus (which uses updatable thunks) yields a lazy machine [**6**].
- CPS-transforming and defunctionalizing the traditional call-by-value monadic evaluator equipped with a monad for a particular effect (state, exceptions, etc., or a combination of effects, including the security technique of stack inspection) yields an abstract machine dedicated to that effect [**7**].
- Defunctionalizing the original evaluation function for the λ-calculus extended with shift and reset yields an extension of the CEK machine for delimited continuations [**41**].
- Defunctionalizing an evaluation function for propositional Prolog with cut yields a traditional abstract machine with two stacks [**46**].

Most of these machines have been independently designed and studied. For example, Felleisen and Friedman's CEK machine takes its roots in Reynolds's work [165, page 196], and during his PhD studies, Schmidt defunctionalized a call-by-name CPS evaluation function, obtaining what is now known as the Krivine machine [319].

[11] *What I cannot create, I do not understand.*
Richard P. Feynman

An abstract machine is not always the defunctionalized counterpart of a CPS program: Not all abstract machines are in defunctionalized form. Examples include William Burge's specification of the SECD machine with the J operator [61], Felleisen's abstract machine for dynamic delimited continuations [163], and the abstract machine corresponding to Olin Shivers and David Fisher's multi-return function calls [327].

What does it mean to be in defunctionalized form?
Nielsen and the author have characterized the target of defunctionalization with one criterion: the inhabitants of the data type representing a function should solely be consumed by the corresponding apply function.

How does one put an abstract machine in defunctionalized form?
This is so far something of an art [**48**, **135**], and the topic of a forthcoming invited talk by the author at the 8th International Conference on Mathematics of Program Construction, MPC 2006 [117].

Is it so desirable for an abstract machine to be in defunctionalized form?
The author believes so since it fits in the functional correspondence, but there is independent evidence: for example, Felleisen's specification of the SECD machine with the J operator [162] differs from the original specification, with no explanation given as to why. Millikin and the author have observed that Felleisen's version is in defunctionalized form, and that the modification corresponds to having a control delimiter in the corresponding evaluation function [**135**]. On the other hand, dynamic delimited continuations have been designed independently of defunctionalization and of CPS [**41**, **47**, **48**, **50**, 159, 237, 238, 323]. Time will tell.

Conclusion: The term "defunctionalization" may not have taken root, according to John Reynolds's assessment in Section 2.2.2, page 40, but defunctionalization nevertheless proves a strikingly valuable tool to connect a compositional evaluation function in continuation-passing style to an abstract machine, i.e., to connect the denotational and operational approaches to computation. In particular, when the evaluation function is compositional (which in the absence of computational reflection is always the case), it should be possible to connect whichever proof technique used to establish an operational property to a proof by structural induction for the corresponding denotational property.

3.2.3 Synergy between the syntactic and the functional correspondences

The syntactic and the functional correspondences synergize, making abstract machines mediate between calculi and evaluation functions. The author illustrated this synergy for arithmetic expressions and for a normalization function in the free monoid [113], and Biernacka and the author illustrated it for calculi of explicit substitutions with computational effects: continuations, delimited continuations, i/o, stack inspection, etc. [**42**]. In each of these cases, starting from the reflexive and transitive closure of a one-step reduction function, one ends with a compositional evaluation function, transiting via several abstract machines.

3.2.4 A constructive corollary of Reynolds's evaluation-order dependence

The goal of this section is to show that the Krivine machine and the CEK machine can in fact be derived from the *same* evaluator for the λ-calculus. This evaluator is the most traditional one for the λ-calculus: it is in direct style, higher-order, compositional, and with an environment (see Figure 4). Were it not already too traditional to need a name, one could refer to it as a "Scott-Tarski" evaluator because, on the one hand, of its recursive data type of values (`expval` in Figure 4), and on the other hand, of its fashion of defining syntactic functions as semantic ones and syntactic applications as semantic ones (`eval` in Figure 4).[12]

As pointed out by Reynolds [304], this traditional evaluator is evaluation-order dependent: if the evaluation order of the defining language is call by name (resp. call by value), the evaluation order of the defined language is also call by name (resp. call by value). The insight here is to specify this evaluation order with the corresponding CPS transformation, i.e., call by name or call by value [295], and to defunctionalize the resulting CPS evaluators.

The abstract syntax of the λ-calculus is implemented as follows:

```
datatype term = IND of int    (* de Bruijn index *)
              | ABS of term
              | APP of term * term
```

A variable is represented by its lexical offset, i.e., its de Bruijn index (starting from 0).

The story is told from Figure 4 to Figure 11.

[12] On the next planet, there was a summer school about gold mining. The little prince walked to one of the students, busy digging a hole together with a robot.
The little prince: *"It's a funny hole you are digging."*
The student: *"You can say that—we are trying to follow a recursive gold vein."*
The little prince: *"A recursive gold vein?"*
The student: *"Yes. It keeps spinning onto itself. Very tricky. But you know, I am glad I came here because I learned a new technique for my robot. By the way, meet my robot. His name is 3CPO. He helps me to dig holes. And with what I have learned here, I have improved him. See his bottom here? It's pointed. Much better to dig recursive veins. OK, pointed 3CPO, let's get back to work!"*
And they resumed digging their hole. The little prince thought that this was quite a peculiar planet. Everybody looks so busy, he thought—except perhaps not all, because there was someone munching a pencil and looking at the clouds drifting in the sky, with a very large stack of papers next to him.
The little prince (curious): *"Hello. You are not digging any hole?"*
The man (pensive): *"Hello, hello. No I don't. I am a theoretician."*
The little prince looked at him blankly.
The theoretician: *"I am from the Tarski institute."*
The little prince: *"And what are you doing?"*
The theoretician, pointing at the stack of papers: *"I am looking into what is gold."*
The little prince: *"So what is gold?"*
The theoretician: *"No, no, not gold. 'Gold'. You need to reify first."*
The little prince: *"What is 'to reify'?"*
The theoretician (looking pleased): *"Right. We also need to do that."*
The little prince (insisting): *"What is 'to reify'?"*
The theoretician (disappointed): *"Oh. You mean that you don't know. Sorry. To reify is to consider someone or something at some distance, as an object."*
The little prince considered.
The little prince (hesitantly): *"At home, I have a rose. My home is at quite a distance, but still I think that she would not want me to consider her as an object."*
The theoretician (stopping to chew his pencil, looking interested, and taking a fresh sheet of paper): *"Maybe so. A rose is a rose, right?"*
The little prince (hastily): *"Perhaps I should let you get back to work. Thank you for your time."*

```
structure Eval0
= struct
    datatype expval = FUNCT of denval -> expval
    withtype denval = expval
         and    env = denval list

    (*  eval : term * env -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (fn v => eval (t, v :: e))
      | eval (APP (t0, t1), e)
        = apply (eval (t0, e), eval (t1, e))
    (*  apply : expval * denval -> expval  *)
    and apply (FUNCT f, a)
        = f a

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

The above evaluator is written in the concrete syntax of Standard ML.

Expressible values, i.e., the result of evaluating an expression, are functions. Denotable values, i.e., the values denoted by variables, coincide with expressible values. (This terminology is due to Strachey [346].) An environment is represented by a list of denotable values.

A variable is evaluated by fetching the corresponding denotable value at its lexical index in the current environment. Evaluating a syntactic function yields a semantic function that, when applied, evaluates the body of the syntactic function in an extension of its lexical environment. Evaluating an application is achieved by applying the value of the operator to the value of the operand.

Overall, the evaluator is compositional (all recursive calls on the right side of the equal sign are made over proper sub-parts of the terms on the left side) and higher order (the domain of expressible values is a function space), with an environment (a list of denotable values).

This evaluator is evaluation-order dependent: as pointed out by Reynolds, whether the term "`eval (t1, e)`" is evaluated at call time or later on (if at all) depends on whether the meta-language follows call by value (as in Standard ML) or, e.g., call by name. In Reynolds's words, the evaluation order of the defining language determines the evaluation order of the defined language.

Figure 4: Traditional evaluation-order dependent evaluator

```
structure Eval1
= struct
    datatype expval = FUNCT of term * env
    withtype denval = expval
         and    env = denval list

    (*  eval : term * env -> expval  *)
    fun eval (IND n, e)
        = List.nth (e, n)
      | eval (ABS t, e)
        = FUNCT (t, e)
      | eval (APP (t0, t1), e)
        = apply (eval (t0, e), eval (t1, e))

    (*  apply : expval * denval -> expval  *)
    and apply (FUNCT (t, e), a)
        = eval (t, a :: e)

    (*  main : term -> expval  *)
    fun main t
        = eval (t, nil)
  end
```

The above evaluator is a first-order counterpart of the evaluator of Figure 4, again in the syntax of Standard ML. It was obtained by defunctionalizing the expressible values 'in-place': only one function abstraction gives rise to inhabitants of

```
Eval0.denval -> Eval0.expval
```

and these inhabitants are already interpreted in `Eval0.apply`. Therefore in the code above, `expval` and `apply` are kept, and only the summand `FUNCT` is changed.

This first-order evaluator is still evaluation-order dependent: whether the term "`eval (t1, e)`" is evaluated at call time or later on (if at all) still depends on whether the meta-language follows call by value or call by name and whether the infix operator `::` evaluates its arguments (which has been a topic of debate in the past [184]).

Figure 5: Evaluator of Figure 4, defunctionalized

```
structure Eval1n
= struct
    datatype expval = FUNCT of term * env
    withtype    ans = expval
         and   cont = expval -> ans
         and denval = cont -> ans
         and    env = denval list

    (*  eval : term * env * cont -> ans  *)
    fun eval (IND n, e, k)
        = List.nth (e, n) k
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn v0 => apply (v0, fn k' => eval (t1, e, k'), k))

    (*  apply : expval * denval * cont -> ans  *)
    and apply (FUNCT (t, e), a, k)
        = eval (t, a :: e, k)

    (*  main : term -> ans  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

The above evaluator is the call-by-name CPS counterpart of the evaluator of Figure 5. Call by name is encoded here because actual parameters are not evaluated at call time and denotable values are (the CPS counterpart of) thunks [208].

Compared to `Eval1.eval`, `Eval1n.eval` takes an extra argument, the continuation. Except for `List.nth`, which is a pure and total function, all calls are tail calls, and in Reynolds's words all sub-expressions are "trivial" to evaluate in the sense that their evaluation incur no effects, including divergence. This evaluator is therefore evaluation-order independent and so one can now say that it is written in Standard ML, not just in the syntax of Standard ML. (One could as well say that it is written in a call-by-name version of ML, for that matter, since it is evaluation-order independent.)

Figure 6: Call-by-name CPS counterpart of Figure 5

```
structure Eval1v
= struct
    datatype expval = FUNCT of term * env
    withtype    ans = expval
         and   cont = expval -> ans
         and denval = expval
         and    env = denval list

    (*  eval : term * env * cont -> ans  *)
    fun eval (IND n, e, k)
        = k (List.nth (e, n))
      | eval (ABS t, e, k)
        = k (FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, fn v0 => eval (t1, e, fn v1 => apply (v0, v1, k)))

    (*  apply : expval * denval * cont -> ans  *)
    and apply (FUNCT (t, e), a, k)
        = eval (t, a :: e, k)

    (*  main : term -> ans  *)
    fun main t
        = eval (t, nil, fn v => v)
  end
```

The above evaluator is the call-by-value CPS counterpart of the evaluator of Figure 5. Call by value is encoded here because actual parameters are evaluated at call time and denotable values are expressible values.

As in Figure 6, `Eval1v.eval` takes an extra argument, the continuation, all calls are tail calls except for `List.nth`, and all sub-expressions are trivial. This evaluator is therefore evaluation-order independent and so one can now say that it is written in Standard ML, not just in the syntax of Standard ML.

Figure 7: Call-by-value CPS counterpart of Figure 5

```
structure Eval1nd
= struct
    datatype expval = FUNCT of term * env
         and denval = THUNK of term * env
    withtype    ans = expval
         and    env = denval list

    datatype cont = CONT1 of term * env * cont
                  | CONT0

    (*  apply_thunk : denval * cont -> ans  *)
    fun apply_thunk (THUNK (t', e'), k')
        = eval (t', e', k')

    (*  apply_cont : cont * expval -> ans  *)
    and apply_cont (CONT1 (t1, e, k), v0)
        = apply (v0, THUNK (t1, e), k)
      | apply_cont (CONT0, v)
        = v

    (*  eval : term * env * cont -> ans  *)
    and eval (IND n, e, k)
        = apply_thunk (List.nth (e, n), k)
      | eval (ABS t, e, k)
        = apply_cont (k, FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT1 (t1, e, k))

    (*  apply : expval * denval * cont -> ans  *)
    and apply (FUNCT (t, e), a, k)
        = eval (t, a :: e, k)

    (*  main : term -> ans  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

The above evaluator is the defunctionalized counterpart of the evaluator of Figure 6. It was obtained by defunctionalizing both the continuations and the denotable values.

Two function abstractions give rise to inhabitants of `Eval1n.cont`: the continuation "`fn v0 => ...`", which has three free variables, and the initial continuation "`fn v => v`", which has none. The data type `cont` and the corresponding `apply_cont` function follow.

One function abstraction gives rise to inhabitants `Eval1n.denval`: the CPS encoding of the thunk "`fn k' => ...`", which has two free variables. The data type `denval` and the corresponding `apply_thunk` function follow.

Figure 8: Defunctionalized counterpart of Figure 6

```
structure Eval1nd'
= struct
    datatype expval = CLOSURE of term * env
    withtype denval = expval
         and    ans = expval
         and    env = denval list
         and   cont = (term * env) list

    (*  eval : term * env * cont -> ans  *)
    fun eval (IND n, e, k)
        = let val (CLOSURE (t', e')) = List.nth (e, n)
          in eval (t', e', k)
          end
      | eval (ABS t, e, (t1, e1) :: k)
        = eval (t, (CLOSURE (t1, e1)) :: e, k)
      | eval (ABS t, e, nil)
        = CLOSURE (t, e)
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, (t1, e) :: k)

    (*  main : term -> ans  *)
    fun main t
        = eval (t, nil, nil)
  end
```

In Figure 8, the data types `Eval1nd.expval` and `Eval1nd.denval` are isomorphic, so they can be merged into one (recursive) data type of closures pairing terms and environments. Also, the data type `cont` is isomorphic to that of lists, so it can be represented as a list of pairs. Finally, all the apply functions are extraneous so they can be inlined. The result coincides with the Krivine machine, which operates on triples consisting of a term, an environment, and a stack of expressible values:

- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t, e]$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t, \mathit{nil}, \mathit{nil}\rangle$
$\langle n, e, s\rangle$	$\Rightarrow$	$\langle t', e', s\rangle$ where $\mathit{nth}(e, n) = [t', e']$
$\langle \lambda t, e, [t_1, e_1] :: s\rangle$	$\Rightarrow$	$\langle t, [t_1, e_1] :: e, s\rangle$
$\langle t_0\, t_1, e, s\rangle$	$\Rightarrow$	$\langle t_0, e, [t_1, e] :: s\rangle$
$\langle \lambda t, e, \mathit{nil}\rangle$	$\Rightarrow$	$[t, e]$

Figure 9: The Krivine machine [90]

```
structure Eval1vd
= struct
    datatype expval = FUNCT of term * env
    withtype    ans = expval
         and denval = expval
         and    env = denval list

    datatype cont = CONT2 of term * env * cont
                  | CONT1 of denval * cont
                  | CONT0

    (*  apply_cont : cont * expval -> ans  *)
    fun apply_cont (CONT2 (t1, e, k), v0)
        = eval (t1, e, CONT1 (v0, k))
      | apply_cont (CONT1 (v0, k), v1)
        = apply (v0, v1, k)
      | apply_cont (CONT0, v)
        = v

    (*  eval : term * env * cont -> ans  *)
    and eval (IND n, e, k)
        = apply_cont (k, List.nth (e, n))
      | eval (ABS t, e, k)
        = apply_cont (k, FUNCT (t, e))
      | eval (APP (t0, t1), e, k)
        = eval (t0, e, CONT2 (t1, e, k))

    and apply (FUNCT (t, e), a, k)
        = eval (t, a :: e, k)

    (*  main : term -> ans  *)
    fun main t
        = eval (t, nil, CONT0)
  end
```

The above evaluator is the defunctionalized counterpart of the evaluator of Figure 7. It was obtained by defunctionalizing the continuations.

Three function abstractions give rise to inhabitants of `Eval1v.cont`: the continuation "`fn (FUNCT (t', e')) => ...`", which has three free variables, the continuation "`fn v1 => ...`", which has two free variables, and the initial continuation "`fn v => v`", which has none. The data type `cont` and the corresponding `apply_cont` function follow.

Inlining `apply` yields the transition functions of the CEK machine (Figure 11).

Figure 10: Defunctionalized counterpart of Figure 7

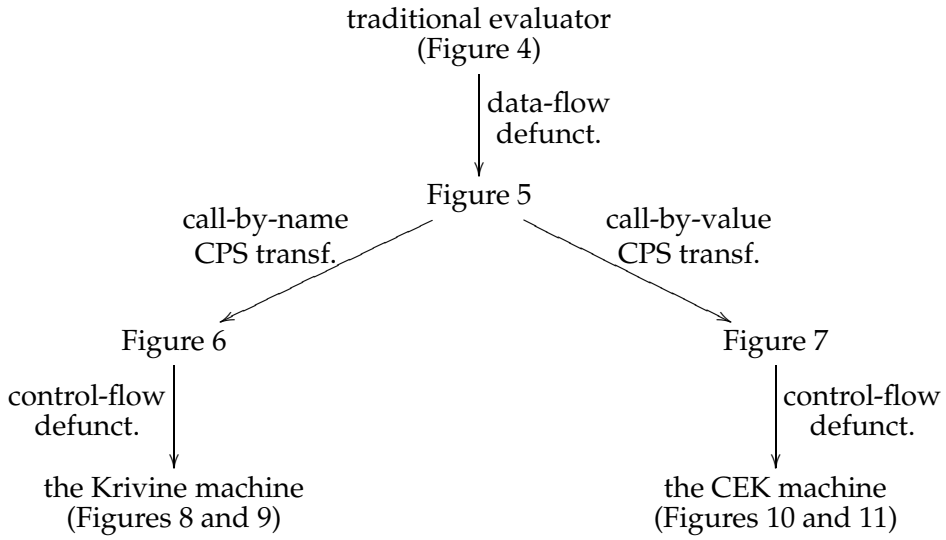
- Source syntax: $t ::= n \mid \lambda t \mid t_0\, t_1$
- Expressible values (closures): $v ::= [t,\ e]$
- Evaluation contexts: $k ::= \texttt{CONT0} \mid \texttt{CONT1}\,(v,\ k) \mid \texttt{CONT2}\,(t,\ e,\ k)$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow_{\mathit{init}}$	$\langle t,\ \mathit{nil},\ \texttt{CONT0}\rangle$
$\langle n,\ e,\ k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\ v\rangle$ where $\mathit{nth}(e,\ n) = v$
$\langle \lambda t,\ e,\ k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle k,\ [t,\ e]\rangle$
$\langle t_0\, t_1,\ e,\ k\rangle$	$\Rightarrow_{\mathit{eval}}$	$\langle t_0,\ e,\ \texttt{CONT2}\,(t_1,\ e,\ k)\rangle$
$\langle \texttt{CONT2}\,(t_1,\ e,\ k),\ v_0\rangle$	$\Rightarrow_{\mathit{apply}}$	$\langle t_1,\ e,\ \texttt{CONT1}\,(v_0,\ k)\rangle$
$\langle \texttt{CONT1}\,([t',\ e'],\ k),\ v_1\rangle$	$\Rightarrow_{\mathit{apply}}$	$\langle t',\ v_1 :: e',\ k\rangle$
$\langle \texttt{CONT0},\ v\rangle$	$\Rightarrow_{\mathit{final}}$	v

The CEK abstract machine consists of two mutually recursive transition functions. The first transition function operates on triples consisting of a term, an environment, and an evaluation context. The second operates on pairs consisting of an evaluation context and an expressible value.

Figure 11: The CEK machine [165]

The moral of the story told from Figure 4 to Figure 11 is that if the CPS transformation is call-by-name, the resulting transition function is that of the Krivine abstract machine, and if the CPS transformation is call-by-value, the resulting transition function is that of the CEK machine:



Reynolds's point was that in general the evaluation order of the defining language, in a definitional interpreter, determines the evaluation order of the defined language if the definitional interpreter is in direct style (and does not use thunks). Ager, Biernacki, Midtgaard, and the author have already shown that a call-by-name interpreter leads one to the Krivine machine and that a call-by-value interpreter leads one to the CEK machine [**5**, Section 2]. But as has been shown from Figure 4 to Figure 11, the Krivine machine and the CEK machine can in fact be derived from the *same* traditional evaluator by letting the CPS transformation encode the evaluation order [206]. In particular, other CPS transformations would lead to other abstract machines, e.g., a lazy one for call by need [**6**, 284].

In any case, whether discovered, invented, or derived, the Krivine abstract machine and the CEK machine have been celebrated independently and in their own right. Yet, as shown here, they are two sides of the same canonical coin.

4 Mixing transformation and execution: Partial evaluation

Partial evaluation is a program-transformation strategy designed to implement Kleene's S_n^m-theorem efficiently [227]. Its goal is to specialize programs. As mentioned in Section 1.1.4, specializing `fold_nat'` in the definition of `power1_with_fold_nat'` yields the definition of `power1`, and specializing `fold_nat'` in the definition of `power4_with_fold_nat'` yields the definition of `power4`. For a simpler example, specializing `power1` with respect to the exponent `10` yields the following residual program:

```
fun power1_x_10 x
    = sqr (mul (x, sqr (sqr (mul (x, sqr 1)))))
```

All the calls to `visit` have been inlined, all the let expressions have been unfolded, and all the conditional expressions have been reduced. With a bit more perspicacity at partial-evaluation time, the expression "`mul (x, sqr 1)`" could be symbolically simplified to "`x`". In any case, applying `power1_x_10` to a base integer raises this integer to the 10th power. Partial evaluation offers a prime example where programs are considered as data objects.

4.1 Background

In the mid 1980's, the partial-evaluation world was brimming with excitement and Copenhagen was its center stage, as Neil Jones, Peter Sestoft, and Harald Søndergaard implemented the first effectively self-applicable partial evaluator, Mix [229, 230]. Neil Jones and Steven Muchnick's concept of *binding times* [228] was the secret weapon to this effectivity. Indeed, Jones introduced the concept of *offline* partial evaluation as a two-stage process: (1) binding-time analysis and (2) specialization. In that setting, effective self-application is obtained by specializing the specializer *after* binding-time analysis instead of specializing both the binding time analysis and the specializer. The whole story is told in Jones's invited paper "Mix ten years after" at PEPM'95 [226].

The author's contribution to partial evaluation at the time was two-fold:

- Charles Consel and the author showed how to apply Mix-style partial evaluation to string matching effectively [80], thereby answering Yoshihiko Futamura's KMP challenge [185].
- Anders Bondorf and the author designed an extensible, automatic, and self-applicable partial evaluator for a first-order subset of Scheme with state-like side effects, Similix [57]. (Bondorf then went on, composing Similix with defunctionalization to make it higher-order [53, 54], implementing a high-performance, state-of-the-art binding-time analysis based on constraints with Jesper Jørgensen [59], improving the specializer with the continuation-based technique of the one-pass CPS transformation [55, 250], and generally maintaining Similix [56].)

4.2 Contribution, impact, and future work

4.2.1 Overall presentations and implementation issues

The author gave general presentations of partial evaluation (1) at a conference with Charles Consel in 1993 [85], (2) at a workshop in 1993 [99], (3) at a spring school at CMU with Charles

Consel and Peter Lee in 1994, (4) at the Marktoberdorf summer school in 1999 [111], and (5) as an encyclopedia article together with John Hatcliff in 2000 [128].

Identifying that the specializer to a large extent interprets binding-time annotations in an offline partial evaluator, Consel and the author have shown how to compile these annotations instead [81]. They have also implemented the first polyvariant partial evaluator operating in parallel [84]. Finally, Karoline Malmkjær, Nevin Heintze, and the author have raised the issue of resource-bounded partial evaluation [129].

4.2.2 Let insertion, CPS, and program analysis

Central to the design of Similix was the desire to correctly handle call unfolding in the presence of effectful computations (including divergence). So for example, specializing an expression such as

```
#1 (42, f x)
```

to `42` or an expression such as

```
(fn y => y + y) (f x)
```

to "`(f x) + (f x)`" was not an option since evaluating "`f x`" might have an effect and therefore this application should not disappear, not be duplicated, and not be reordered. So instead, these expressions were specialized by inserting a residual let expression, resulting in

```
let val y = f x
in 42
end
```

and

```
let val y = f x
in y + y
end
```

respectively.

Unfortunately, the residual let expressions stand in the way of further specialization in general. The author tentatively proposed to use CPS to bypass them [98], by processing the body of the let expression with the continuation of this let expression, as in the CPS transformation [122]. It then turned out that a pre-CPS transformation (or just considering source programs in CPS to start with) worked wonders for specializing interpreters [83]; Consel and the author thus studied the effect of a pre-CPS transformation over binding-time analysis [82]. Furthermore, the continuation-based handling of let expressions also benefits the specializer itself, both with an explicit continuation as shown in Bondorf's work [55] and in direct style with shift and reset, as shown by Lawall and the author [250].

Eventually, Hatcliff and the author found out that both let insertion and the CPS gymnastics to improve binding times are actually accounted for by Eugenio Moggi's computational metalanguage and monadic reductions [207]. Furthermore, Karoline Malmkjær, Jens Palsberg, and the author established that what the CPS transformation does to control flow in terms of binding-time improvement, eta expansion does to data flow [133, 134]. The conjunction of both paved the way to type-directed partial evaluation (Section 5).

Meanwhile, however, Amr Sabry and Matthias Felleisen questioned the impact of the CPS transformation over flow analysis [313]. They showed that for their constant-propagation algorithm, analyzing a direct-style program and analyzing its CPS counterpart yields incomparable results: CPS might increase precision by duplicating continuations and it might decrease precision by confusing return points. Palsberg and Wand, in their later study [288], developed a CPS transformation of flow information and supported the conclusion that the extra precision enabled by a CPS transformation is due to the duplication of the analysis.

However, Sabry and Felleisen's analysis (1) depends on the order in which the source program is traversed and (2) is duplicated over conditional branches—two properties that led to the conclusion that the CPS transformation does not preserve the result of constant propagation. Instead, Damian and the author considered off-the-shelf constraint-based analyses that do not depend on the order in which constraints are solved and where the analyses are not duplicated over conditional branches: a monovariant control-flow analysis [283], a traditional monovariant binding-time analysis, and a monovariant binding-time analysis for continuation-based partial evaluation. Through a systematic construction of the CPS counterpart of flow information, they found that control-flow analysis is insensitive to the CPS transformation, that the CPS transformation does improve binding times for traditional partial evaluation, and that the binding-time analysis for continuation-based partial evaluation is insensitive to the CPS transformation [92, **95**]. They also pointed out that prior to the introduction of continuations [313, Footnote 2], let flattening provides an improvement for binding-time analysis whereas let 'deepening' provides an improvement for region inference [**95**, Section 8.1.1], and concluded on the need for more robust program analyses that are less vulnerable to syntactic accidents in source programs.

4.2.3 Jones optimality

On the road towards self-applying a partial evaluator, Neil Jones identified the following useful stepping stone: the partial evaluator should be able to entirely remove interpretive overheads. So in particular, specializing a (direct-style) self-interpreter with respect to a source program should yield (an alpha-renamed version of) this source program. A partial evaluator satisfying this property has since been said to be 'Jones-optimal' [266].

While partial evaluators for untyped languages such as lambda-Mix are provably Jones-optimal [330], the residual type tags stand in the way for typed languages, requiring new developments in partial-evaluation technology for the typed world. During the 1990's, obtaining Jones-optimality in a typed setting has proved an abundant source of inspiration in partial evaluation and related areas, perhaps most notably with John Hughes's type specialization [224].

The author made two attempts at obtaining Jones optimality in a typed setting. The first attempt used type coercions and required source programs to be well typed [106]. But it fell short of the mark since the starting point was not a self-interpreter. The second attempt, reported jointly with López [131], hinged on a simple representation shift of the specialized version of a typed lambda-interpreter: reading the type-tagged residual programs as higher-order abstract syntax. With this representation shift, ordinary partial evaluation is already Jones optimal.

4.2.4 String matching

Consel and the author's initial work on partial evaluation for string matching [80] was not reverse-engineered from the Knuth-Morris-Pratt string matcher [239]. Instead, it was forward-designed based on Jones's notion of binding times in offline partial evaluation, and it was a wonderful surprise to see how the residual programs implemented the second phase of the KMP [80]. In the wake of this wonderful surprise, at the turn of the 1990's, the author boldly conjectured that while a left-to-right traversal gave rise to a family of Knuth-Morris-Pratt linear-time string matchers, a right-to-left traversal similarly should give rise to a family of Boyer-Moore linear-time string matchers, the key issue being to monitor and prune positive and negative information at partial-evaluation time. The overall approach is summarized in a joint book chapter with Torben Amtoft, Charles Consel, and Karoline Malmkjær [13].

4.2.5 String matching, revisited

W. C. Fields: *"There is something about a lady's boudoir..."*
Mae West: *"How do you know?"*

How does one know indeed that specializing a source string matcher with respect to a string gives rise to a residual matcher that behaves like a particular, independently known, string matcher? At the turn of the 2000's, Mads Sig Ager, Henning Korsholm Rohde, and the author took as a criterion *the sequence of indices at which characters are compared*, and proved that the initial claim of obtaining the Knuth-Morris-Pratt string matcher [80] was formally correct [**8**]. The formalization, however, is not lightweight enough to repeat for every possible source matcher and every corresponding string matcher [70]. As an alternative, Rohde has automated a test suite with which to compare sequences of indices [309, 310]. He used this test suite to verify the bold conjecture about left-to-right and right-to-left string matchers, and found it overly general: in actuality, one must both

- creatively use bounded static variation to generate positive information, and
- selectively restrict the propagation of static information, whether positive or negative,

to avoid ending in a maze of twisty little matchers, all alike. Besides making it possible to assess published claims, Rohde's test suite can also be used to obtain new results: for example, it supported solving the long-standing open problem of obtaining the Boyer-Moore string-matching algorithm by partial evaluation [**144**].

4.2.6 Specialization time (and space)

At the turn of the 2000's, Grobauer and Lawall formalized the time complexity and the size of specialized string matchers [199]. Their work shed a concrete light on whether one could hope to obtain not just the second half of the Knuth-Morris-Pratt string matcher, i.e., a linear traversal of the text, but also the first half, i.e., a linear-time generation of a 'next' table [239]. Ager, Rohde, and the author have therefore stated requirements on a partial evaluator for it to specialize a string matcher in linear time and construct a residual program that runs in linear time [**9**].

5 Mixing transformation and execution: Type-directed partial evaluation

Type-directed partial evaluation is a partial-evaluation technique where to specialize a program, one executes it in a 'residualizing' environment. This section reviews the background of this technique, presents it step by step, and puts it into perspective.

5.1 Background

The starting point of type-directed partial evaluation was the one-pass CPS transformation [121, 122, 123]. As developed in Section 2.3, the usual CPS transformation operates in two passes: the first one generates CPS code, and the second one performs administrative reductions. The idea of operating in one pass was to perform the administrative reductions at transformation time. To this end, the binding times of the CPS transformation were separated by two-level eta-expansion [123, Section 2]. The binding-time improvement technique of eta-expansion originates there, and so does the technique of relocating the continuation of a let expression in its body at transformation time.

Karoline Malmkjær, Jens Palsberg, and the author then jointly studied two-level eta-expansion both for binding-time separation and for binding-time improvement [133, 134, 287]. The goal was, instead of circumventing the coarse approximations of existing binding-time analyses, to design better binding-time analyses that would not require any source eta-expansion.

It was in that context, in March 1995, that Thomas Streicher gave a talk at DAIMI on "reduction-free normalization" [11], where he presented the very same two-level eta-redexes. It then became suddenly clear to the author that rather than making two-level eta-expansion fit an existing offline partial evaluator, the partial evaluator should be *replaced* by a two-level eta-expander, and type-directed partial evaluation was born [**103**].

5.2 Contribution, impact, and future work

5.2.1 Two-level programming

At the turn of the 1990's, there was serious ongoing work on two-level programming languages, involving the design of binding-time typing systems and of dedicated two-level code processors [282, 286, 371]. The author's approach was considerably more minimalistic [111]: it aimed at constraining the static level of a two-level language as little as possible, ultimately letting it espouse the typing discipline of an existing functional language. Doing so makes it possible to use a pre-existing programming language (Scheme or ML) for writing two-level programs while standing on the shoulders of the quasiquotation giants [31].

For offline partial evaluation, the benefit is obvious: the typing discipline of the host language ensures the binding-time discipline of the source program, and its execution environment carries out its specialization. No particular knowledge of binding-time analysis and its subtleties is required: one only needs to know the typing system of one's programming language. And other than quasiquotation, no other particular knowledge of program specialization is required either: one only needs to know one's programming language.

The rest of this section shifts to using Scheme, which is dynamically typed but offers linguistic support for quasiquotation [235].

5.2.2 Two-level programming in Scheme

For example, consider the following definition of the exponentiation function:

```
(define power
  (lambda (x n)
    (letrec ([visit (lambda (n)
                      (if (zero? n)
                          1
                          (visit-positive n)))]
             [visit-positive (lambda (m)
                               (if (even? m)
                                   (sqr (visit-positive (quotient m 2)))
                                   (mul (sqr (visit (quotient m 2))) x)))])
      (visit n))))
```

In an environment where `sqr` and `mul` implement a squaring function and a multiplication function and where `zero?`, `even?`, and `quotient` are bound as usual, evaluating "`(power 2 10)`" yields `1024`.

Given an exponent, the following 'generating extension' [266] constructs the representation of a function that raises its argument to that exponent:

```
(define powergen
  (lambda (n)
    `(lambda (x)
       ,(letrec ([visit (lambda (n)
                          (if (zero? n)
                              '1
                              (visit-positive n)))]
                 [visit-positive (lambda (m)
                                   (if (even? m)
                                       `(sqr ,(visit-positive (quotient m 2)))
                                       `(mul (sqr ,(visit (quotient m 2))) x)))])
          (visit n)))))
```

And indeed, evaluating "`(powergen 10)`" yields

```
(lambda (x)
  (sqr (mul (sqr (sqr (mul (sqr 1) x))) x)))
```

i.e., the representation of a function that raises its argument to the 10th power, as in Section 4, page 69.

5.2.3 Two-level programming in Scheme, revisited

This is not a pipe.
Magritte

There are alternatives to writing the generating extension above. For example, instead of performing a squaring operation, `sqr` could construct a representation of this squaring operation, and similarly for `mul`. To this end, `sqr` and `mul` can be defined as simple code-generating functions:

```
(define sqr
  (lambda (i)
    `(sqr ,i)))

(define mul
  (lambda (i j)
    `(mul ,i ,j)))
```

In such an environment, evaluating "`(power '2 10)`" yields

```
(sqr (mul (sqr (sqr (mul (sqr 1) 2))) 2))
```

i.e., a representation of how to compute the result instead of the result, which is a binding-time shift. In particular, evaluating "`` `(lambda (x) ,(power 'x 10)) ``" yields

```
(lambda (x)
  (sqr (mul (sqr (sqr (mul (sqr 1) x))) x)))
```

i.e., the representation of a function that raises its argument to the 10th power, just as above, but using the original definition of `power` (and the code-generating versions of `sqr` and `mul`) instead of the corresponding generating extension.

That quasiquotation makes it possible to construct residual programs is well known (it is even its raison d'être). The thesis here is two-fold:

What: Localizing quasiquotation in code-generating functions makes it possible to achieve partial evaluation over ordinarily staged programs.

How: Code-generating functions can be defined by two-level eta-expansion.

The 'what' part is known: for example, it is at the basis of Andy Berlin's partial evaluator for numerical programs [37, 38]. The 'how' part is specific to binding-time coercions as in the author's work with Malmkjær and Palsberg [133, 134] and to higher-order one-pass code generation as in the author's work with Filinski [122, 123]:

From representation to value: reflection. In the definition of `sqr` just above,

```
(lambda (i) `(sqr ,i))
```

is a two-level eta-redex. It is two-level because the introduction (the lambda-abstraction) is static and the elimination (the application) is dynamic. ('Static' stands for normal computation, and 'dynamic' stands for code generation.)

Its one-level counterpart, i.e.,

```
(lambda (i) (sqr i))
```

is an eta-redex.

From value to representation: reification. In the residualization of `power` above,

```
`(lambda (x) ,(power 'x 10))
```

is also a two-level term. It is two-level because the lambda-abstraction is dynamic and the application is static. ('Dynamic' stands for code generation, and 'static' stands for normal computation.)

Its one-level counterpart, i.e.,

```
(lambda (x) (power x 10))
```

is an ordinary Scheme function.

5.2.4 Towards type-directed partial evaluation

Let us systematize reflection and reification based on two-level eta-expansion:

Reflection.

Given the name of a unary function, `reflect1` yields a unary function that, given the representation of an argument, yields the representation of a call of the named function to this argument (and likewise for `reflect2` and binary functions):

```
(define reflect1    ;;; exp -> (exp -> exp)
  (lambda (f)
    (lambda (x)
      `(,f ,x))))

(define reflect2    ;;; exp -> (exp * exp -> exp)
  (lambda (f)
    (lambda (x y)
      `(,f ,x ,y))))
```

Reification.

Given a polymorphic unary function, `reify1` yields a representation of this function:

```
(define reify1    ;;; (exp -> exp) -> exp
  (lambda (f)
    `(lambda (x)
       ,(f 'x))))
```

Thus equipped, one can define `sqr` and `mul` as the result of reflecting upon their name, as on page 75,

```
(define sqr (reflect1 'sqr))

(define mul (reflect2 'mul))
```

and one can define a function mapping an exponent to the representation of a function that raises its argument to this exponent:

```
(define power_x    ;;; int -> exp
  (lambda (n)
    (reify1 (lambda (x)
              (power x n)))))
```

For example, evaluating "`(power_x 10)`" yields

```
(lambda (x)
  (sqr (mul (sqr (sqr (mul (sqr 1) x))) x)))
```

as on page 75, though in practice, one uses fresh names in residual programs.

5.2.5 Type-directed partial evaluation

This is a factory making normalized representations of pipes.

First, let us settle on a representation of types, using a simple record facility (`define-record` and `case-record`), as common in Scheme [183], and writing a parser from a S-expression to a parsed type (for simplicity, only unary functions are considered for now):

```
(define-record (Base name))
(define-record (Prod first-type second-type))
(define-record (Func domain-type range-type))
```

So for example, evaluating "`(parse-type '(a -> b -> a * b))`" yields the abstract representation of the concrete type `(a -> b -> a * b)`:

```
(Func (Base "a") (Func (Base "b") (Prod (Base "a") (Base "b"))))
```

Second, let us define reification and reflection with two mutually recursive functions, following the structure of a given type. We are then in position to define a mapping from a polymorphic value and its type to its representation.

```
(define residualize       ;;; two-level eta-expansion
  (lambda (v t)
    (letrec ([reify       ;;; type-directed map from value to representation
              (lambda (t v)
                (case-record t
                  [(Base)
                   v]
                  [(Prod t1 t2)
                   `(cons ,(reify t1 (car v)) ,(reify t2 (cdr v)))]
                  [(Func t1 t2)
                   (let ([x1 (gensym!)])
                     `(lambda (,x1)
                        ,(reify t2 (v (reflect t1 x1)))))]))]
             [reflect     ;;; type-directed map from representation to value
              (lambda (t e)
                (case-record t
                  [(Base)
                   e]
                  [(Prod t1 t2)
                   (cons (reflect t1 `(car ,e)) (reflect t2 `(cdr ,e)))]
                  [(Func t1 t2)
                   (lambda (v1)
                     (reflect t2 `(,e ,(reify t1 v1))))]))])
      (reify (parse-type t) v))))
```

In an environment where `sqr` and `mul` are defined as in Section 5.2.4, evaluating "`(residualize (lambda (x) (power x 10)) '(a -> a))`" yields

```
(lambda (x33)
  (sqr (mul (sqr (sqr (mul (sqr 1) x33))) x33)))
```

as on page 76, but with a fresh name (`x33`) instead of a constant one (`x`).

To sum up [**103**, **107**]:

1. programs that specialize well with a (monovariant) offline partial evaluator can be expressed with quasiquotation in a way that simulates the partial-evaluation process;
2. binding-time annotations can be represented with quasiquotation in a way such that running the quasiquoted programs achieves the specialization process (a.k.a. the 'cogen approach');
3. quasiquotation can be factored out of source programs and localized in type-directed definitions of reification and reflection.

With this final insight, one can stage the power function by factoring `sqr` and `mul` [266]:

```
(define make-power
  ;;; int -> (a -> a) * (a * a -> a) -> a -> a
  (lambda (n)
    (lambda (sqr mul)
      (lambda (x)
        (letrec ([visit (lambda (n)
                          (if (zero? n)
                              1
                              (visit-positive n)))]
                 [visit-positive (lambda (m)
                                   (if (even? m)
                                       (sqr (visit-positive (quotient m 2)))
                                       (mul (sqr (visit (quotient m 2))) x)))])
          (visit n))))))
```

To define the standard exponentiation function, `make-power` is instantiated with an exponent, the standard squaring and multiplication functions, and a base integer:

```
(define power
  (lambda (x n)
    (((make-power n) (lambda (i) (* i i)) *) x)))
```

And to construct a specialized version of the exponentiation function with respect to a given exponent, e.g., 10, "`(make-power 10)`" is residualized at type

```
(((a -> a) ((a a) => a)) => a -> a)
```

where "`=>`" marks the type of an uncurried function. The result reads as follows:

```
(lambda (f40 f41)
  (lambda (i42)
    (f40 (f41 (f40 (f40 (f41 (f40 1) i42))) i42))))
```

Given the standard squaring and multiplication functions, this function yields another function that, given a base integer, raises it to the 10th power. Its definition is, however, very much unreadable.

5.2.6 Type-directed partial evaluation with control over residual names

What's in a name?
Juliet

Foremost, the residual programs are unreadable because repeated residualization yields non-identical results. For example, residualizing again "`(make-power 10)`" at the same type as above is likely to read as something like

```
(lambda (f43 f44)
  (lambda (i45)
    (f43 (f44 (f43 (f43 (f44 (f43 1) i45))) i45))))
```

which is alpha-equivalent but not telling to the human eye. A first fix is to make residual names more uniform, either by re-initializing the generator of fresh names at the beginning of residualization or by adopting a more uniform naming scheme such as de Bruijn levels. This way, repeated residualization is guaranteed to yield textually the same result.

Uniformity, however, is not sufficient. So in the author's implementation, the user can declare types with *name stubs* for naming residual variables. Residualizing "`(make-power 10)`" at type

```
((((a -> a) alias "sqr") (((a a) => a) alias "mul")) => (a with "x") -> a)
```

where `with` specifies a name stub for `gensym!` and `alias` specifies the name of any variable of this type (caveat emptor), then yields the following residual program:

```
(lambda (sqr mul)
  (lambda (x0)
    (sqr (mul (sqr (sqr (mul (sqr 1) x0))) x0))))
```

This program is readable.[13] (Changing "`with`" into "`alias`" in the type above would yield "`x`" instead of "`x0`" in the residual program.)

Getting back to Streicher's talk on "reduction-free normalization" [11], the residualization function of page 77, which the author views as an economical, factored way to specialize staged programs, is known in other circles as a *normalization function* mapping pure λ-terms to their long βη-normal form (more about that in Section 6). And indeed, it can be used to decompile the value of closed λ-terms into their normal form, witness the following classical combinators [30]. Let us first define them in Scheme:

```
> (define S (lambda (f) (lambda (g) (lambda (x) ((f x) (g x))))))
> (define K (lambda (x) (lambda (y) x)))
> (define I ((S K) K))
> (define B ((S (K S)) K))
> (define C ((S (K ((S (K ((S S) (K K)))) K))) S))
> (define W ((S S) (K I)))
>
```

[13] Question: *"How can you tell when you've reached Lisp Enlightenment?"*
Answer: *"The parentheses disappear."*
(found at <http://en.wikiquote.org/wiki/Lisp_programming_language#Parentheses>)

Given their type, one can decompile `B`, `C`, `W`, and `K` into their normal form (with the default naming and also with names specified in the types):

```
> (residualize B '((b -> c) -> (a -> b) -> a -> c))
(lambda (f0) (lambda (f1) (lambda (a2) (f0 (f1 a2)))))
> (residualize B '(((b -> c) alias "f") ->
                   ((a -> b) alias "g") ->
                         (a alias "x") -> c))
(lambda (f) (lambda (g) (lambda (x) (f (g x)))))
> (residualize C '((b -> a -> c) -> a -> b -> c))
(lambda (x0) (lambda (a1) (lambda (b2) ((x0 b2) a1))))
> (residualize C '(((b -> a -> c) alias "f") ->
                               (a alias "x") ->
                               (b alias "y") -> c))
(lambda (f) (lambda (x) (lambda (y) ((f y) x))))
> (residualize W '((a -> a -> b) -> a -> b))
(lambda (x0) (lambda (a1) ((x0 a1) a1)))
> (residualize W '(((a -> a -> b) alias "f") -> (a alias "x") -> b))
(lambda (f) (lambda (x) ((f x) x)))
> (residualize K '(a -> b -> a))
(lambda (a0) (lambda (b1) a0))
> (residualize K '((a alias "x") -> (b alias "y") -> a))
(lambda (x) (lambda (y) x))
>
```

So the areas of partial evaluation and of normalization overlap—an observation that has been formally substantiated since by Dybjer and Filinski [158, 172].

In any case, the reduction strategy embodied in the residualization function is normal order, so it is unsuited for specializing call-by-value programs. Indeed the normal-order strategy gives everything that was not wanted in Similix (page 70):

```
> (residualize (lambda (f x)
                 (car (cons 42 (f x))))
               '((((a -> b) alias "f") (a alias "x")) => int))
(lambda (f x)
  42)
> (residualize (lambda (add f x)
                 ((lambda (y) (add y y)) (f x)))
               '(((((a a) => a) alias "add")
                     ((a -> a) alias "f")
                           (a alias "x")) => a))
(lambda (add f x)
  (add (f x) (f x)))
>
```

In the first interaction, the application "`(f x)`" has disappeared from the residual program, and in the second one, it has been duplicated.

A mechanism would be to only consider CPS programs since they are insensitive to the evaluation order. A solution would be to do the same as in Similix: let insertion. Let us describe this solution.

5.2.7 Type-directed partial evaluation with let insertion

First, the representation of types is adjusted with name stubs and `gensym!` is replaced by a new procedure, `elaborate-new-name!`, to generate a fresh name for a variable of a certain type based on the stub of that type. A new type constructor is also added, that of procedures, i.e., functions with effects:

```
(define-record (Base name stub))
(define-record (Prod type type stub))
(define-record (Func type type stub))
(define-record (Proc type type stub))
```

On page 77, code duplication arises because of reflection at function type: the whole residual function call is returned. The author's solution is to use delimited continuations (Section 1.4.2): the code-generation continuation is captured, a let expression naming the residual call is manufactured, and the continuation is restored with this name to construct the body of this let expression. In other words, instead of having

```
(lambda (v1)
  (reflect t2 `(,e ,(reify t1 v1))))
```

on page 77, one writes:

```
(lambda (v1)
  (let ([q2 (elaborate-new-name! t2)])
    (shift (lambda (k)
             `(let ([,q2 (,e ,(reify t1 v1))])
                ,(reset (lambda ()
                          (k (reflect t2 q2)))))))))
```

The call-by-value mapping from a polymorphic value and its type to its representation reads as follows:

```
(define residualize       ;;; two-level eta-expansion with let insertion
  (lambda (v t)
    (letrec ([reify       ;;; type-directed map from value to representation
              (lambda (t v)
                (case-record t
                  [(Base name stub)
                   v]
                  [(Prod t1 t2 stub)
                   `(cons ,(reify t1 (car v)) ,(reify t2 (cdr v)))]
                  [(Func t1 t2 stub)
                   (let ([x1 (elaborate-new-name! t1)])
                     `(lambda (,x1)
                        ,(reify t2 (v (reflect t1 x1)))))]
                  [(Proc t1 t2 stub)
                   (let ([x1 (elaborate-new-name! t1)])
                     `(lambda (,x1)
                        ,(reset (lambda ()
                                  (reify t2 (v (reflect t1 x1))))))))]))]
```

```
            [reflect     ;;; type-directed map from representation to value
             (lambda (t e)
               (case-record t
                 [(Base name stub)
                  e]
                 [(Prod t1 t2 stub)
                  (cons (reflect t1 `(car ,e)) (reflect t2 `(cdr ,e)))]
                 [(Func t1 t2 stub)
                  (lambda (v1)
                    (reflect t2 `(,e ,(reify t1 v1))))]
                 [(Proc t1 t2 stub)
                  (lambda (v1)
                    (let ([q2 (elaborate-new-name! t2)])
                      (shift (lambda (k)
                               `(let ([,q2 (,e ,(reify t1 v1))])
                                  ,(reset (lambda ()
                                            (k (reflect t2 q2))))))))]))])
      (begin
        (reset-gensym!)
        (reify (parse-type t) v)))))
```

Control is delimited at the outset of each dynamic context, and abstracted when reflecting at procedure type. As a result, residualizing "`(make-power 10)`" at type

```
((((a -!> a) alias "sqr") (((a a) =!> a) alias "mul")) => (a alias "x") -> a)
```

where `-!>` and `=!>` respectively declare a unary and an uncurried procedure, yields the following residual program:

```
(lambda (sqr mul)
  (lambda (x0)
    (let ([i1 (sqr 1)])
      (let ([i2 (mul i1 x0)])
        (let ([i3 (sqr i2)])
          (let ([i4 (sqr i3)])
            (let ([i5 (mul i4 x0)])
              (let ([i6 (sqr i5)])
                i6)))))))))
```

Compared to the readable residual program on page 79, all the intermediate results of this one are named and their computation is sequentialized.

In practice, the construction of residual let expressions probes its operands and yields the following (tail-call optimized) residual program:

```
(lambda (sqr mul)
  (lambda (x0)
    (let* ([i1 (sqr 1)]
           [i2 (mul i1 x0)]
           [i3 (sqr i2)]
           [i4 (sqr i3)]
           [i5 (mul i4 x0)])
      (sqr i5))))
```

In particular, let us go back to the examples of pages 70 and 80, with an effectful type this time:

```
> (residualize (lambda (f x)
                 (car (cons 42 (f x))))
               '((((a -!> b) alias "f") (a alias "x")) => int))
(lambda (f x)
  (let* ([b0 (f x)])
    42))

> (residualize (lambda (add f x)
                 ((lambda (y) (add y y)) (f x)))
               '(((((a a) => a) alias "add")
                    ((a -!> a) alias "f")
                            (a alias "x")) => a))
(lambda (add f x)
  (let* ([i0 (f x)])
    (add i0 i0)))
>
```

In both residual programs, the application "`(f x)`" is named, and therefore it does not disappear, is not duplicated, and is not reordered, as in Similix.

The part about naming and let insertion was presented at a Dagstuhl meeting in 1996 [102]. Filinski has later shown that, just as let insertion and the CPS gymnastics to improve binding times in Similix are accounted for by Moggi's computational metalanguage and monadic reductions [207], type-directed partial evaluation with let insertion can also be formally characterized as a monadic normalization function [174].

5.2.8 Type-directed partial evaluation with sums

Besides let insertion, delimited continuations also make it possible to handle sum types [**103**, **107**]. For example, reflecting upon a boolean expression is achieved by capturing the current delimited continuation, successively applying it to true and false, which yields a residual consequent and a residual alternative, and returning a residual conditional expression with these two branches. For example, residualizing

```
(lambda (b)
  (+ 1 (if b 20 30)))
```

at type "`(bool -> a)`" yields

```
(lambda (b0)
  (if b0 21 31))
```

where the delimited context "`[(+ 1 [.])]`" has been captured and restored twice, once with `20` and yielding `21`, and once with `30` and yielding `31`. A residual conditional expression has then grouped both branches.

Vincent Balat and the author have experimented with limiting the duplication of residual contexts using a memo-table [27]. Balat then went on farther using dynamic delimited control operators instead of shift and reset, and obtained tighter normal forms [24, 25].

5.2.9 Online type-directed partial evaluation

Type-directed partial evaluation can be extended with predefined functions that probe their arguments for possible simplification [105]. For example, the following definitions of `sqr` and `mul` exploit algebraic properties of squaring and of multiplication: `0` is absorbant and `1` is neutral.

```
(define conceptualize
  (lambda (e t)
    (reflect (parse-type t) e)))

(define sqr
  (lambda (i)
    (case i
      [(0 1) i]
      [else  ((conceptualize 'sqr '(int -> int)) i)])))

(define mul
  (lambda (x y)
    (case x
      [(0)  0]
      [(1)  y]
      [else (case y
              [(0)  0]
              [(1)  x]
              [else ((conceptualize 'mul '((int int) => int)) x y)])])))
```

Residualizing "`((make-power 10) sqr mul)`" at type "`(int -> int)`" then yields

```
(lambda (x0)
  (sqr (mul (sqr (sqr x0)) x0)))
```

where "`(mul (sqr 1) x0)`" has been simplified to `x0` at residualization time, compared to the residual programs of pages 76 and 79.

Online type-directed partial evaluation synergizes with let insertion. For example, if one declares "`(int -!> int)`" and "`((int int) =!> int)`" in the definitions of `sqr` and `mul` above, residualizing "`((make-power 10) sqr mul)`" at type "`(int -> int)`" yields

```
(lambda (x0)
  (let* ([i1 (sqr x0)]
         [i2 (sqr i1)]
         [i3 (mul i2 x0)])
    (sqr i3)))
```

where "`(sqr 1)`" and "`(mul 1 x0)`" have been simplified to `x0` at residualization time, compared to the residual program on page 82. In addition, compared to the residual code just above, all the intermediate results are named and their computation is sequentialized.

Filinski's monadic formalization also accounts for online type-directed partial evaluation [174, Section 5].

5.2.10 Offline type-directed partial evaluation and the base case

Often, it is a good idea to unfold the test for the base case instead of making a blind recursive call. Such an unfolding makes it possible to carry out algebraic simplifications prior to partial evaluation, i.e., offline, instead of probing for them during partial evaluation, i.e., online.

For example, one can inline `visit` in the definition of `make-power`, on page 78:

```
(define make-power-unfolded
  ;;; int -> (a -> a) * (a * a -> a) -> a -> a
  (lambda (n)
    (lambda (sqr mul)
      (lambda (x)
        (letrec ([visit-positive
                  (lambda (m)
                    (if (even? m)
                        (sqr (visit-positive (quotient m 2)))
                        (let ([n (quotient m 2)])
                          (if (zero? n)
                              (mul (sqr 1) x)
                              (mul (sqr (visit-positive n)) x)))))])
          (if (zero? n)
              1
              (visit-positive n)))))))
```

Simplifying the consequent "`(mul (sqr 1) x)`" into `x` eliminates the need for `mul` and `sqr` to probe their arguments for possible simplification at residualization time.

5.2.11 Type-directed partial evaluation and interpreter specialization

Type-directed partial evaluation realizes semantics-directed compiling into the intermediate language of long βη monadic normal forms [206, 268]: the author illustrated this realization for vanilla imperative languages [102, **103**], for action notation together with Morten Rhiger [142], and for an Algol-like language together with René Vestergaard [149]. William Harrison also used type-directed partial evaluation for semantics-directed compiling [204].

As to functional languages, the type transformation induced by an interpreter can be used to residualize the meaning of a typed expression [101]. For example, if a λ-interpreter is in CPS, applying it to an expression e of type `t` yields a value whose type is the CPS counterpart of `t`; residualizing this value at that type yields the CPS counterpart of e in normal form.

5.2.12 Run-time code generation

Balat and the author experimented with translating and loading the output of type-directed partial evaluation into byte code for the OCaml virtual machine [26]. Rhiger deforested the representation of normal forms to directly generate byte code, as in Michael Sperber and Peter Thiemann's work in traditional syntax-directed partial evaluation [334]; he applied it in Jason Hickey's Ensemble system [308]. Grobauer, Rhiger, and the author implemented a JIT compiler for a subset of the goal-directed programming language Icon [**125**]. The output of the compiler is comparable to that of Todd Proebsting's compiler [301], not by any other design than because its output is in normal form.

5.2.13 Related work and alternatives

In traditional syntax-directed partial evaluation, quasiquotation is mostly used as an alternative to self-application to obtain a compiler generator (a.k.a. 'cogen') [51, 58, 220, 356]. Simon Helsen and Peter Thiemann pointed out the similarity between type-directed partial-evaluation and the cogen approach [210] (see also [**107**, Section 6]). Rowan Davies and Frank Pfenning presented logical foundations for binding-time analysis and staged computation [151, 152]. Walid Taha developed a typed approach to nested quasiquotation for multi-stage programming [352].

At the turn of the 1990's, at MIT [37, 38], Andy Berlin developed a partial evaluator that was based on running Scheme code with instrumented primitive procedures probing their actual parameters in search for simplifications, as on page 84 but without two-level eta-expansion [105, Section 4]. Also at the turn of the 1990's, at Indiana University [188], Mayer Goldberg studied Gödelization in Scheme [189], pioneering the first functions that would decompile proper combinators into their source code and eventually extending Gödelization to all strongly normalizing combinators.

At the turn of the 2000's, Grobauer and Yang restated type-directed partial evaluation to make it self-applicable, and they implemented the second Futamura projection [200]. To this end, they needed to monomorphize occurrences of shift and reset, which also improved efficiency. Eijiro Sumii and Naoki Kobayashi improved efficiency even more with a state-based, rather than control-based, implementation of let insertion [349]. Thiemann proved the equivalence of both implementations using a type system with effects [359].

6 Mixing transformation and execution: normalization by evaluation

I learned very early the difference
between knowing the name of something
and knowing something.
Richard P. Feynman [168, The Making of a Scientist]

The term "normalization by evaluation" is due to Helmut Schwichtenberg in 1998 [35], but the phenomenon it refers to drew attention earlier in several computer-science areas, under at least one other name: "reduction-free normalization" [11, 89]. The key idea is to define

1. an *evaluation* function from a syntactic domain of terms to a semantic domain of values, mapping syntactically equivalent terms to the same semantic value, and
2. an inverse *reification* function from values to terms.

The composition of these two functions yields terms in normal form and acts as a *normalization function*. In contrast to the usual reduction-based approaches to normalization, a normalization function is "reduction free" because

what: it is based on a notion of undirected equivalence, not on a notion of directed reduction; and

how: all the normalization work occurs in the meta-language. For example, if one characterizes terms in normal form with a specialized data type, then the *type* of the normalization function (from terms to terms in normal form) reflects its normalizing nature, with the type inferencer of the implementation language acting in effect as a theorem prover that establish this partial correctness [143].

In 1998, Peter Dybjer and the author organized an APPSEM workshop on normalization by evaluation [119]. Rarely has the author attended such a thematically focused meeting—it was impressive to see how the same normalization function for the simply typed λ-calculus could arise in so many distinct areas. A comprehensive review of normalization by evaluation can be found in Dybjer and Filinski's lecture notes at the APPSEM summer school in 2000 [158]. The rest of this section reviews what has happened since.

Normalization by evaluation: a silver bullet? First of all, though, it should be stressed that normalization by evaluation is no silver bullet. For example, at the 2000 APPSEM summer school, in his joint lecture with Dybjer and Filinski, the author has presented a fully parameterized version of Coquand and Dybjer's normalization function for Gödel's system T in combinatory form [86]: the normalization function could be run in call-by-value, call-by-name, and call-by-need mode, and the normal forms could be constructed eagerly or lazily. While most of the time this normalization function outperforms an ordinary reduction-based normalizer, for each combination (call by value, call by name, etc.), the author exhibited a term for which this particular combination did worse than the others. The issue is the same as in Reynolds's work on definitional interpreters [304]: that of the evaluation order of the meta-language. For a simple example, reducing a non-strict combinator such as K—whether in the language (i.e., in a reduction-based manner) or in the meta-language (i.e., in a reduction-free manner)—is best done in a non-strict way.

Domain of applications of normalization by evaluation: Most of the work on normalization by evaluation has taken place in the setting of the λ-calculus:

- typed, as in Filinski's work on the computational λ-calculus [174] and in Thorsten Altenkirch and Tarmo Uustalu's work on $\lambda^{\rightarrow 2}$ [12]; and
- untyped, as in Mayer Goldberg's [188] and in Torben Mogensen's work in Scheme [265], in Klaus Aehlig and Felix Joachimski's operational approach [2], and in Filinski and Rohde's denotational account [175].

Combinatory logic is a close second. Until recently, the only other example of normalization by evaluation that is not related to λ-terms or combinators appears to be the free monoid [39, 86, 236]. Biernacka, Biernacki, and the author have proposed a new example: a hierarchical language of propositions [**41**, Section 6].

Abstract machines for strong reduction:

One theoretician's strongly reducing abstract machine
is another programmer's defunctionalized normalization function.

Other approaches exist for implementing strong normalization, notably dedicated abstract machines [88, 194, 256]. In the light of the functional correspondence (Section 3.2.2), the author and his students have found that these abstract machines are defunctionalized versions of normalization functions [4, 263].

Weak head normalization functions: In 1993 [34], Ulrich Berger re-constructed the original normalization function from his joint work with Helmut Schwichtenberg [36] as a program extracted from a normalization proof due to Tait. Recently, Małgorzata Biernacka, Kristian Støvring, and the author have repeated the experiment for a *weak head* normalization proof [44]. They wanted to see how close such an extracted normalization function is to an evaluation function such as the one in Figure 4, page 60—and the answer is: far, due to the pairs of syntactic terms and semantic values induced by glueing. More work is needed here.

One-pass transformations as instances of normalization by evaluation: To close the circle, let us come back to the one-pass CPS transformation, since it was the starting point of type-directed partial evaluation. In principle, it is an instance of normalization by evaluation in Martin-Löf's sense since it uses the meta-language to carry out the administrative reductions towards administrative normal form. Millikin has recently shown that so is it in practice as well, since he reconstructed the one-pass CPS transformation by (1) composing Plotkin's original CPS transformation with a normalization function for the λ-calculus and (2) deforesting the intermediate administratively unreduced CPS terms [261].

7 Conclusion and perspectives

I am still confused,
but at a much higher level.
Enrico Fermi

The goal of this essay was to convey to the reader that manipulating programs as data objects can shed a constructive light on their design. Over the years, the concerns presented here have provided a fertile research ground for PhD students here in Aarhus [3, 40, 45, 92, 198, 260, 262, 281, 308, 309, 343, 380] as well as, earlier on, in the US [28, 188, 205, 248] and in France [78]. Since their graduation, all of the latter students and one third of the former students in Aarhus have pursued a career into academia, while the other two thirds have moved to research and to the industry.

It has long been my personal view
that the separation of practical and theoretical work
is artificial and injurious.
Christopher Strachey

Acknowledgments: This essay was completed while visiting Neil Jones's TOPPS group at DIKU and Patrick Cousot's research group at the ENS.

The author is grateful to Karoline Malmkjær, Jan Midtgaard, Kevin Millikin, and Kristian Støvring for their feedback. Special thanks to Andrzej Filinski and Julia Lawall for their insightful comments and their scientific influence not just on this essay but on virtually all of the 32 enclosed articles.

"The Little Prince and the Programmer" (Footnote 5, page 32) was written by Karoline Malmkjær and the author in 1987.

8 Bibliographic references

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Massachusetts, 1985.

[2] Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 14:587–611, 2004.

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

[4] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, March 2003.

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

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

[7] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the technical report BRICS RS-04-28 (December 2004). √

[8] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On obtaining Knuth, Morris, and Pratt's string matcher by partial evaluation. In Wei-Ngan Chin, editor, *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 32–46, Aizu, Japan, September 2002. ACM Press. Extended version available as the technical report BRICS RS-02-32 (July 2002). √

[9] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast partial evaluation of pattern matching in strings. *ACM Transactions on Programming Languages and Systems*, 2006. To appear. Available as the technical report BRICS RS-04-40 (December 2004). A preliminary version was presented at the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2003). √

[10] Alex Aiken, editor. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. ACM Press.

[11] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[12] Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming, 7th International Symposium, FLOPS 2004*, number 2998 in Lecture Notes in Computer Science, pages 260–275, Nara, Japan, April 2004. Springer-Verlag.

[13] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. In Mogensen et al. [267], pages 332–357.

[14] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, Texas, January 1989. ACM Press.

[15] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Małuszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[16] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[17] Andrew W. Appel and Zhong Shao. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, 2000.

[18] Zena M. Ariola, Hugo Herbelin, and Amr Sabry. A type-theoretic foundation of continuations and prompts. In Fisher [178], pages 40–53.

[19] Kenichi Asai. *The Reflective Language Black*. PhD thesis, Department of Information Science, Faculty of Science, University of Tokyo, Tokyo, Japan, December 1996.

[20] Kenichi Asai. Online partial evaluation for shift and reset. In Peter Thiemann, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2002)*, SIGPLAN Notices, Vol. 37, No 3, pages 19–30, Portland, Oregon, March 2002. ACM Press.

[21] Kenichi Asai. Offline partial evaluation for shift and reset. In Nevin Heintze and Peter Sestoft, editors, *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2004)*, pages 3–14, Verona, Italy, August 2003. ACM Press.

[22] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation: For a better understanding of reflective languages. *Lisp and Symbolic Computation*, 9(2/3):203–241, 1996.

[23] Lennart Augustsson. A compiler for Lazy ML. In Steele [340], pages 218–227.

[24] Vincent Balat. *Une étude des sommes fortes: isomorphismes et formes normales*. PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, December 2002.

[25] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, pages 64–76, Venice, Italy, January 2004. ACM Press.

[26] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

[27] Vincent Balat and Olivier Danvy. Memoization in type-directed partial evaluation. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 2002 ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, number 2487 in Lecture Notes in Computer Science, pages 78–92, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.

[28] Anindya Banerjee. *The Semantics and Implementation of Bindings in Higher-Order Programming Languages*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, July 1995.

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

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

[31] Alan Bawden. Quasiquotation in Lisp. In Danvy [108], pages 4–12. Available online at <http://www.brics.dk/~pepm99/programme.html>.

[32] Michael Beeler, R. William Gosper, and Rich Schroeppel. HAKMEM. AI Memo 239, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1972. Available at <http://home.pipeline.com/~hbaker1/hakmem/>.

[33] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In Tofte [360], pages 25–37.

[34] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[35] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

[36] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[37] Andrew A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Master's thesis, MIT Artificial Intelligence Laboratory, July 1989. Technical report 1144.

[38] Andrew A. Berlin. Partial evaluation applied to numerical computation. In Wand [370], pages 139–150.

[39] Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95*, number 1158 in Lecture Notes in Computer Science, pages 47–61, Torino, Italy, June 1995. Springer-Verlag.

[40] Małgorzata Biernacka. *A Derivational Approach to the Operational Semantics of Functional Languages*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, January 2006.

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

[42] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. Technical Report BRICS RS-05-38, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2005. Accepted for publication in Theoretical Computer Science (March 2006). √

[43] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the technical report BRICS RS-06-3 (February 2006). √

[44] Małgorzata Biernacka, Olivier Danvy, and Kristian Støvring. Program extraction from proofs of weak head normalization. In Martin Escardó, editor, *Proceedings of the 21st Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, Birmingham, UK, May 2005. Elsevier Science Publishers. Extended version available as the technical report BRICS RS-05-12.

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

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

[47] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited √ control. *Journal of Functional Programming*, 2006. To appear. Available as the technical report BRICS RS-05-25 (August 2005).

[48] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style √ for dynamic delimited continuations. Technical Report BRICS RS-05-16, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 2005.

[49] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the dynamic extent of delimited continuations. *Information Processing Letters*, 96(1):7–17, 2005. Extended version available as the technical report BRICS RS-05-36.

[50] Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents √ of delimited continuations. *Science of Computer Programming*, 2006. To appear. Available as the technical report BRICS RS-05-36 (December 2005).

[51] Lars Birkedal and Morten Welinder. Handwriting program generator generators. In Hermenegildo and Penjam [213], pages 198–214.

[52] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

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

[54] Anders Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.

[55] Anders Bondorf. Improving binding times without explicit cps-conversion. In Clinger [72], pages 1–10.

[56] Anders Bondorf. Similix 5.1 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, May 1993. Included in the Similix 5.1 distribution.

[57] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[58] Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 1–10, Orlando, Florida, June 1994.

[59] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

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

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

[62] Rod M. Burstall. Writing search algorithms in functional form. In Donald Michie, editor, *Machine Intelligence*, volume 5, pages 373–385. Edinburgh University Press, 1969.

[63] Luca Cardelli. Compiling a functional language. In Steele [340], pages 208–217.

[64] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[65] Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

[66] Robert (Corky) Cartwright, editor. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991. ACM Press.

[67] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Smolka [333], pages 56–71.

[68] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In Pierce [293], pages 241–253.

[69] Eugene Charniak, Christopher Riesbeck, and Drew McDermott. *Artificial Intelligence Programming*. Lawrence Earlbaum Associates, 1980.

[70] Christian Charras and Thierry Lecroq. Exact string matching algorithms. `<http://www-igm.univ-mlv.fr/~lecroq/string/>`, 1997.

[71] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, February 1993.

[72] William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.

[73] William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, 1985.

[74] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In Talcott [353], pages 128–139.

[75] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In Cartwright [65], pages 124–131.

[76] William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999. Journal version of [75].

[77] William D. Clinger. Proper tail recursion and space efficiency. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 174–185, Montréal, Canada, June 1998. ACM Press.

[78] Charles Consel. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université Pierre et Marie Curie (Paris VI), Paris, France, June 1989.

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

[80] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[81] Charles Consel and Olivier Danvy. From interpreting to compiling binding times. In Neil D. Jones, editor, *Proceedings of the Third European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 88–105, Copenhagen, Denmark, May 1990. Springer-Verlag.

[82] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [223], pages 496–519.

[83] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Cartwright [66], pages 14–24.

[84] Charles Consel and Olivier Danvy. Partial evaluation in parallel. *Lisp and Symbolic Computation*, 5(4):315–330, 1993.

[85] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [191], pages 493–501.

[86] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[87] Pierre Crégut. An abstract machine for lambda-terms normalization. In Wand [370], pages 333–340.

[88] Pierre Crégut. Strongly reducing variants of the Krivine abstract machine. In Danvy [118]. To appear. Journal version of [87].

[89] Djordje Čubrić, Peter Dybjer, and Philip J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

[90] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[91] Ron K. Cytron, editor. *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, Las Vegas, Nevada, June 1997. ACM Press.

[92] Daniel Damian. *On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-5.

[93] Daniel Damian and Olivier Danvy. A simple CPS transformation of control-flow information. *Logic Journal of the IGPL*, 10(5):501–515, 2002.

[94] Daniel Damian and Olivier Danvy. CPS transformation of flow information, part II: Administrative reductions. *Journal of Functional Programming*, 13(5):925–934, 2003. √

[95] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. *Journal of Functional Programming*, 13(5):867–904, 2003. A preliminary version was presented at the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000). √

[96] Olivier Danvy. Memory allocation and higher-order functions. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 241–252, Saint-Paul, Minnesota, June 1987. ACM Press.

[97] Olivier Danvy. On listing list prefixes. *LISP Pointers*, 2(3-4):42–46, January 1989.

[98] Olivier Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37(6):315–322, March 1991. Extended version available as the technical report 303, Computer Science Department, Indiana University, January 1990.

[99] Olivier Danvy. Évaluation partielle: principes, pratique, et perspectives. Journées Francophones des Langages Applicatifs, Annecy, France. Exposé invité, January 1993.

[100] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994. A preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992). √

[101] Olivier Danvy. Décompilation de lambda-interprètes. In Guy Lapalme and Christian Queinnec, editors, *JFLA 96 – Journées francophones des langages applicatifs*, volume 15 of *Collection Didactique*, pages 133–146, Val-Morin, Québec, January 1996. INRIA.

[102] Olivier Danvy. Pragmatics of type-directed partial evaluation. Research Report BRICS RS-96-15, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1996.

[103] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press. √

[104] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998. Extended version available as the technical report BRICS RS-98-12. √

[105] Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998. World Scientific.

[106] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 908–917, Aalborg, Denmark, July 1998. Springer-Verlag.

[107] Olivier Danvy. Type-directed partial evaluation. In Hatcliff et al. [209], pages 367–411. √

[108] Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1999)*, Technical report BRICS NS-99-1, University of Aarhus, San Antonio, Texas, January 1999.

[109] Olivier Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In Aart Middeldorp and Taisuke Sato, editors, *Functional and Logic Programming, 4th International Symposium, FLOPS 1999*, number 1722 in Lecture Notes in Computer Science, pages 241–250, Tsukuba, Japan, November 1999. Springer-Verlag. √

[110] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Smolka [333], pages 88–103. √

[111] Olivier Danvy. Programming techniques for partial evaluation. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, *Foundations of Secure Computation*, NATO Science series, pages 287–318. IOS Press Ohmsha, 2000.

[112] Olivier Danvy. A new one-pass transformation into monadic normal form. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 77–89, Warsaw, Poland, April 2003. Springer-Verlag. √

[113] Olivier Danvy. From reduction-based to reduction-free normalization. In Sergio Antoy and Yoshihito Toyama, editors, *Proceedings of the Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, number 124 in Electronic Notes in Theoretical Computer Science, pages 79–100, Aachen, Germany, May 2004. Elsevier Science. Invited talk.

[114] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In Hayo Thielecke, editor, *Proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (CW'04)*, Technical report CSR-04-1, Department of Computer Science, Queen Mary's College, pages 13–23, Venice, Italy, January 2004. Invited talk. √

[115] Olivier Danvy. A rational deconstruction of Landin's SECD machine. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, number 3474 in Lecture Notes in Computer Science, pages 52–71, Lübeck, Germany, September 2004. Springer-Verlag. Recipient of the 2004 Peter Landin prize. Extended version available as the technical report BRICS RS-03-33 (October 2003). √

[116] Olivier Danvy. Sur un exemple de Patrick Greussay. Research Report BRICS RS-04-41, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2004.

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

[118] Olivier Danvy, editor. *Special Issue on the Krivine Abstract Machine*, Higher-Order and Symbolic Computation. Springer, 2006. In preparation.

[119] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, University of Aarhus. Available online at `<http://www.brics.dk/~nbe98/programme.html>`.

[120] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*, pages 19–31, Paris, France, September 1999.

[121] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. DIKU Rapport 89/12, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, July 1989.

[122] Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [370], pages 151–160.

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

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

[125] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53–73, 2002. Extended version available as the technical report BRICS RS-01-29 (July 2001). A preliminary version was presented at the Second International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001). √

[126] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[127] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[128] Olivier Danvy and John Hatcliff. Partial evaluation. In Anthony Ralston, Edwin Reilly, and David Hemmendinger, editors, *Encyclopedia of Computer Science (Fourth Edition)*, pages 1370–1371. Macmillan Reference/Grove Dictionaries, 2000.

[129] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[130] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [72], pages 299–310.

[131] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, number 2618 in Lecture Notes in Computer Science, pages 335–347, Warsaw, Poland, April 2003. Springer-Verlag.

[132] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in a reflective tower. In Cartwright [65], pages 327–341.

[133] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995. A preliminary version was presented at the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1994).

[134] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[135] Olivier Danvy and Kevin Millikin. A rational deconstruction of Landin's J operator. In Greg J. Michaelson, editor, *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05*, number ? in Lecture Notes in Computer Science, Dublin, Ireland, September 2005. Springer-Verlag. To appear. Extended version available as the technical report BRICS RS-06-4 (February 2006). √

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

[137] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[138] Olivier Danvy and Lasse R. Nielsen. A first-Order one-pass CPS transformation. *Theoretical Computer Science*, 308(1-3):239–257, 2003. A preliminary version was presented at the Fifth International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002). √

[139] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. A preliminary version appears in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[140] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. *Information Processing Letters*, 94(5):217–224, 2005. Extended version available as the research report BRICS RS-04-39.

[141] Olivier Danvy and Frank Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1995.

[142] Olivier Danvy and Morten Rhiger. Compiling actions by type-directed partial evaluation. In Jüri Vain, editor, *Proceedings of the 9th Nordic Workshop on Programming Theory*, Estonian Academy of Sciences Series, Tallinn, Estonia, October 1997. Extended version available as the research report BRICS RS-98-13.

[143] Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

[144] Olivier Danvy and Henning Korsholm Rohde. On obtaining the Boyer-Moore string-matching algorithm by partial evaluation. *Information Processing Letters*, 2006. To appear. Available as the technical report BRICS RS-05-29 (September 2005). √

[145] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 90–106, Amsterdam, The Netherlands, June 1997. ACM Press.

[146] Olivier Danvy and Ulrik P. Schultz. Lambda-dropping: transforming Recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1-2):243–287, 2000. √

[147] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. In Zhenjiang Hu and Mario Rodriguez-Artalejo, editors, *Functional and Logic Programming, 6th International Symposium, FLOPS 2002*, number 2441 in Lecture Notes in Computer Science, pages 134–151, Aizu, Japan, September 2002. Springer-Verlag. Extended version available as the technical report BRICS RS-03-26.

[148] Olivier Danvy and Ulrik P. Schultz. Lambda-lifting in quadratic time. *Journal of Functional and Logic Programming*, 10(1), July 2004. Available online at `<http://danae.uni-muenster.de/lehre/kuchen/JFLP/>`. A preliminary version was presented at the Sixth International Symposium on Functional and Logic Programming (FLOPS 2002). √

[149] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS RS-96-13.

[150] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[151] Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[152] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001. A preliminary version was presented at the Twenty-Third Annual ACM Symposium on Principles of Programming Languages (POPL 1996).

[153] Jim des Rivières and Brian C. Smith. The implementation of procedurally reflective languages. In Steele [340], pages 331–347.

[154] Mary S. Van Deusen and Zvi Galil, editors. *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985. ACM Press.

[155] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[156] Scott Draves. Implementing bit-addressing with specialization. In Tofte [360], pages 239–250.

[157] Bruce F. Duba, Robert Harper, and David B. MacQueen. Typing first-class continuations in ML. In Cartwright [66], pages 163–173.

[158] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[159] R. Kent Dybvig, Simon Peyton-Jones, and Amr Sabry. A monadic framework for subcontinuations. Technical Report 615, Computer Science Department, Indiana University, Bloomington, Indiana, June 2005.

[160] Leonidas Fegaras. lambda-DB. Available online at `<http://lambda.uta.edu/lambda-DB/manual/>`, 1999-2001.

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

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

[163] Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

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

[165] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[166] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

[167] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [65], pages 52–62.

[168] Richard P. Feynman. *"What do you care what other people think?"*. Bantam Books, 1989.

[169] Andrzej Filinski. Representing monads. In Boehm [52], pages 446–457.

[170] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

[171] Andrzej Filinski. Representing layered monads. In Aiken [10], pages 175–188.

[172] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.

[173] Andrzej Filinski. An extensional CPS transform (preliminary report). In Amr Sabry, editor, *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*, Technical report 545, Computer Science Department, Indiana University, pages 41–46, London, England, January 2001.

[174] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[175] Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004*, number 2987 in Lecture Notes in Computer Science, pages 167–181, Barcelona, Spain, April 2002. Springer-Verlag.

[176] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, 13(3):509–543, 2003.

[177] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, 1993. Available at <http://www.brics.dk/~hosc/vol06/03-fischer.html>. A preliminary version was presented at the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[178] Kathleen Fisher, editor. *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 39, No. 9, Snowbird, Utah, September 2004. ACM Press.

[179] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In McKinley [257], pages 502–514.

[180] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In Deusen and Galil [154], pages 245–254.

[181] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In Steele [340], pages 348–355.

[182] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[183] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, second edition*. The MIT Press, 2001.

[184] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In S. Michaelson and Robin Milner, editors, *Third International Colloquium on Automata, Languages, and Programming*, pages 257–284. Edinburgh University Press, Edinburgh, Scotland, July 1976.

[185] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[186] Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Peyton Jones [291], pages 271–282.

[187] Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming*. Palgrave Macmillan, 2003.

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

[189] Mayer Goldberg. Gödelization in the λ-calculus. *Information Processing Letters*, 75(1-2):13–16, 2000.

[190] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[191] Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[192] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In Martin S. Feather and Michael Goedicke, editors, *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 211–222, Coronado Island, San Diego, California, USA, November 2001. IEEE Computer Society.

[193] Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In David Sands, editor, *Proceedings of the Tenth European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 122–136, Genova, Italy, April 2001. Springer-Verlag.

[194] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Peyton Jones [291], pages 235–246.

[195] Patrick Greussay. *Contribution à la définition interprétative et à l'implémentation des λ-langages*. Thèse d'état, Université de Paris VII, Paris, France, 1977.

[196] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[197] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983.

[198] Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-6.

[199] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[200] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

[201] Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

[202] G. H. Hardy. *A Mathematician's Apology, reprinted with a foreword by C. P. Snow*. Canto. Cambridge University Press, 1993.

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

[204] William L. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2001.

[205] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[206] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [52], pages 458–471.

[207] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997. Extended version available as the technical report BRICS RS-96-34.

[208] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997. Extended version available as the technical report BRICS RS-97-7.

[209] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, July 1998. Springer-Verlag.

[210] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In Jieh Hsiang and Atsushi Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998. Springer-Verlag.

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

[212] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In Fisher [178], pages 16–27.

[213] Manuel Hermenegildo and Jaan Penjam, editors. *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, Madrid, Spain, September 1994. Springer-Verlag.

[214] Carl Hewitt, Peter Bishop, Richard Steiger, Irene Greif, Brian Smith, Todd Matson, and Roger Hale. Behavioral semantics of nonrecursive control structures. In Bernard Robinet, editor, *Symposium on Programming*, number 19 in Lecture Notes in Computer Science, pages 385–407, Paris, France, April 1974. Springer-Verlag.

[215] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.

[216] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[217] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[218] Ralf Hinze. Deriving backtracking monad transformers. In Philip Wadler, editor, *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 35, No. 9, pages 186–197, Montréal, Canada, September 2000. ACM Press.

[219] Ralf Hinze. Formatting: a class act. *Journal of Functional Programming*, 13(5):935–944, 2003.

[220] Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

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

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

[223] John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[224] John Hughes. Type specialisation for the lambda calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 183–215, Dagstuhl, Germany, February 1996. Springer-Verlag.

[225] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jouannaud [231], pages 190–203.

[226] Neil D. Jones. Mix ten years after. In William L. Scherlis, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 24–38, La Jolla, California, June 1995. ACM Press.

[227] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `<http://www.dina.kvl.dk/~sestoft/pebook/>`.

[228] Neil D. Jones and Steven S. Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[229] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, number 202 in Lecture Notes in Computer Science, pages 124–140, Dijon, France, May 1985. Springer-Verlag.

[230] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[231] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, Nancy, France, September 1985. Springer-Verlag.

[232] Yukiyoshi Kameyama. Axioms for delimited continuations in the CPS hierarchy. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 442–457, Karpacz, Poland, September 2004. Springer.

[233] Yukiyoshi Kameyama. Axioms for control operators in the CPS hierarchy. *Higher-Order and Symbolic Computation*, 2006. To appear.

[234] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.

[235] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[236] Yoshiki Kinoshita. A bicategorical analysis of E-categories. *Mathematica Japonica*, 47(1):157–169, 1998.

[237] Oleg Kiselyov. How to remove a dynamic prompt: Static and dynamic delimited continuation operators are equally expressible. Technical Report 611, Computer Science Department, Indiana University, Bloomington, Indiana, March 2005.

[238] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers. In Pierce [293], pages 192–203.

[239] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[240] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM Press.

[241] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In Stuart I. Feldman, editor, *Proceedings of the 1986 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 21, No 7, pages 219–233, Palo Alto, California, June 1986. ACM Press.

[242] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In Danvy [118]. To appear. Available online at `<http://www.pps.jussieu.fr/~krivine/>`.

[243] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[244] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[245] Peter J. Landin. A generalization of jumps and labels. Research report, UNIVAC Systems Programming Research, 1965. Reprinted in Higher-Order and Symbolic Computation 11(2):125–143, 1998, with a foreword [354].

[246] Peter J. Landin. Getting rid of labels. Research report, UNIVAC Systems Programming, July 1965.

[247] Peter J. Landin. Histories of discoveries of continuations: Belles-lettres with equivocal tenses. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW'97)*, Technical report BRICS NS-96-13, University of Aarhus, pages 1:1–9, Paris, France, January 1997.

[248] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

[249] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Graham [191], pages 124–136.

[250] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Talcott [353], pages 227–238.

[251] Xavier Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, February 1990.

[252] Harry Mairson. Outline of a proof theory of parametricity. In Hughes [223], pages 313–327.

[253] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In John Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Rapport de recherche N° 2265, INRIA*, pages 112–119, Orlando, Florida, June 1994.

[254] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

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

[256] Clement L. McGowan. The correctness of a modified SECD machine. In *Proceedings of the Second Annual ACM Symposium in the Theory of Computing*, pages 149–157, Northampton, Massachusetts, May 1970.

[257] Kathryn S. McKinley, editor. *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979–1999, A Selection*. ACM Press, 2004.

[258] Chris Mellish and Steve Hardy. Integrating Prolog in the POPLOG environment. In John A. Campbell, editor, *Implementations of PROLOG*, pages 147–162. Ellis Horwood, 1984.

[259] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer-Verlag.

[260] Jan Midtgaard. From implicit to explicit contexts in operational semantics. Progress report, BRICS PhD School, University of Aarhus, December 2004.

[261] Kevin Millikin. A new approach to one-pass transformations. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP 2005)*, pages 252–264, Tallinn, Estonia, September 2005. Institute of Cybernetics at Tallinn Technical University. Granted the best student-paper award of TFP 2005.

[262] Kevin Millikin. On obtaining one-pass transformations via fusion. Progress report, BRICS PhD School, University of Aarhus, June 2005.

[263] Kevin Millikin. A rational reconstruction of three abstract machines for strong normalization. In Neil Jones, editor, *Proceedings of the 17th Nordic Workshop on Programming Theory*, pages 97–99, Copenhagen, Denmark, October 2005. DIKU, Computer Science Department, University of Copenhagen.

[264] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.

[265] Torben Æ. Mogensen. Gödelization in the untyped lambda-calculus. In Danvy [108], pages 19–24.

[266] Torben Æ. Mogensen. Glossary for partial evaluation and related topics. *Higher-Order and Symbolic Computation*, 13(4):355–368, 2000.

[267] Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[268] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[269] Jean-François Monin. Proof pearl: From concrete to functional unparsing. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004*, number 3223 in Lecture Notes in Computer Science, pages 217–224, Park City, Utah, September 2004. Springer.

[270] Marco T. Morazán and Barbara Mucha. Reducing extraneous parameters during lambda lifting. In *Implementation and Application of Functional Languages, 17th International Workshop, IFL'05, preliminary proceedings*, Dublin, Ireland, September 2005. `<https://www.cs.tcd.ie/ifl05/>`.

[271] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Hermenegildo and Penjam [213], pages 182–197.

[272] Akimasa Morihata, Kazuhiko Kakehi, Zhenjiang Hu, and Masato Takeichi. Reversing iterations: IO swapping leads you there and back again. Mathematical Engineering Technical Reports METR 2005-11, Department of Mathematical Informatics, Graduate School of Information Science and Technology, the University of Tokyo, Tokyo, Japan, May 2005.

[273] F. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993. Reprinted from a manuscript dated 1970.

[274] Peter D. Mosses. A foreword to 'Fundamental concepts in programming languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

[275] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

[276] Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the First ACM SIGPLAN Workshop on Continuations (CW'92)*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

[277] Peter Naur. The design of the GIER ALGOL compiler, part I. *BIT*, 3:124–140, 1963.

[278] Peter Møller Neergaard. *Complexity Aspects of Programming Language Design—From Logspace to Elementary Time via Proofnets and Intersection Types*. PhD thesis, Mitchom School of Computer Science, Brandeis University, Waltham, Massachusetts, October 2004.

[279] Lasse R. Nielsen. A denotational investigation of defunctionalization. Progress report (superseded by [280]), BRICS PhD School, University of Aarhus, June 1999.

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

[281] Lasse R. Nielsen. *A study of defunctionalization and continuation-passing style*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-7.

[282] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[283] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[284] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, 7(3/4):57–82, 1994.

[285] Bruno C. d. S. Oliveira and Jeremy Gibbons. Typecase: a design pattern for type-indexed functions. In Daan Leijen, editor, *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 98–109, Tallinn, Estonia, September 2005.

[286] Jens Palsberg. Correctness of binding-time analysis. *Journal of Functional Programming*, 3(3):347–363, 1993.

[287] Jens Palsberg. Eta-redexes in partial evaluation. In Hatcliff et al. [209], pages 356–366.

[288] Jens Palsberg and Mitchell Wand. CPS transformation of flow information. *Journal of Functional Programming*, 13(5):905–923, 2003.

[289] John Allan Paulos. *I think, therefore I laugh*. Columbia University Press, New York, 2000.

[290] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In Pierce [293], pages 216–227.

[291] Simon Peyton Jones, editor. *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 37, No. 9, Pittsburgh, Pennsylvania, September 2002. ACM Press.

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

[293] Benjamin Pierce, editor. *Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, SIGPLAN Notices, Vol. 40, No. 9, Tallinn, Estonia, September 2005. ACM Press.

[294] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

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

[296] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[297] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2001. Technical Report CMU-CS-01-152.

[298] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In Jean-Yves Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, number 1581 in Lecture Notes in Computer Science, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag.

[299] Jeff Polakow and Frank Pfenning. Properties of terms in continuation passing style in an ordered logical framework. In Joëlle Despeyroux, editor, *Workshop on Logical Frameworks and Meta-Languages (LFM 2000)*, Santa Barbara, California, June 2000.

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

[301] Todd A. Proebsting. Simple translation of goal-directed evaluation. In Cytron [91], pages 1–6.

[302] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Cartwright [66], pages 174–184.

[303] John Reppy. Optimizing nested loops using local CPS conversion. *Higher-Order and Symbolic Computation*, 15(2/3):161–180, 2002.

[304] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363–397, 1998, with a foreword [306].

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

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

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

[308] Morten Rhiger. *Higher-Order Program Generation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, July 2001. BRICS DS-01-4.

[309] Henning Korsholm Rohde. *Formal Aspects of Partial Evaluation*. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, December 2005.

[310] Henning Korsholm Rohde. Measuring the propagation of information in partial evaluation. Research Report BRICS RS-05-26, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, August 2005.

[311] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[312] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[313] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In Vivek Sarkar, editor, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 29, No 6, pages 1–12, Orlando, Florida, June 1994. ACM Press.

[314] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(6):916–941, 1997. A preliminary version was presented at the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP 1996).

[315] Erik Sandewall. An early use of continuations and partial evaluation for compiling rules written in FOPC. *Higher-Order and Symbolic Computation*, 12(1):105–113, 1999.

[316] David A. Schmidt. State transition machines for lambda calculus expressions. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, pages 415–440, Aarhus, Denmark, 1980. Springer-Verlag.

[317] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[318] David A. Schmidt. *The Structure of Typed Programming Languages*. The MIT Press, 1994.

[319] David A. Schmidt. State-transition machines for lambda-calculus expressions. In Danvy [118]. To appear. Journal version of [316].

[320] Ulrik P. Schultz. Implicit and explicit aspects of scope and block structure. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1997.

[321] Peter Sestoft. Demonstrating lambda calculus reduction. In Mogensen et al. [267], pages 420–435.

[322] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the 2004 ACM SIGPLAN Workshop on Scheme and Functional Programming*, Technical report TR600, Computer Science Department, Indiana University, Snowbird, Utah, September 2004.

[323] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 2006. Journal version of [322]. To appear.

[324] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Special Issue: PLI Workshops*, ACM SIGPLAN Notices Vol. 37 No. 12, pages 60–75, December 2002.

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

[326] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned (with retrospective). In McKinley [257], pages 257–269.

[327] Olin Shivers and David Fisher. Multi-return function call. In Fisher [178], pages 79–89.

[328] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[329] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [370], pages 161–175.

[330] Sebastian Skalberg. Mechanical proof of the optimality of a partial evaluator. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, February 1999.

[331] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT, Cambridge, Massachusetts, January 1982. MIT-LCS-TR-272.

[332] Brian C. Smith. Reflection and semantics in Lisp. In Ken Kennedy, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 1984. ACM Press.

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

[334] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Cytron [91], pages 215–225.

[335] Michael Spivey and Silvija Seres. Combinators for logic programming. In Gibbons and de Moor [187], pages 177–199.

[336] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

[337] Guy L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976.

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

[339] Guy L. Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. In Patrick Henry Winston and Richard Henry Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2, pages 399–431. The MIT Press, 1979.

[340] Guy L. Steele Jr., editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[341] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.

[342] Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[343] Kristian Støvring. Normalization and simple types. Progress report, BRICS PhD School, University of Aarhus, June 2005.

[344] Kristian Støvring. Higher-order beta matching with solutions in long beta-eta normal form. *Nordic Journal of Computing*, 2006. To appear.

[345] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[346] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000, with a foreword [274].

[347] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [365].

[348] Eijiro Sumii. An implementation of transparent migration on standard Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Technical Report 00-368, Rice University, pages 61–64, Montréal, Canada, September 2000.

[349] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[350] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998, with a foreword [351].

[351] Gerald J. Sussman and Guy L. Steele Jr. The first report on Scheme revisited. *Higher-Order and Symbolic Computation*, 11(4):399–404, 1998.

[352] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1999. CSE-99-TH-002.

[353] Carolyn L. Talcott, editor. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[354] Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[355] Hayo Thielecke. Answer type polymorphism in call-by-name continuation passing. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, number 2986 in Lecture Notes in Computer Science, pages 279–293, Barcelona, Spain, April 2004. Springer-Verlag.

[356] Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, Philadelphia, Pennsylvania, May 1996. ACM Press.

[357] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[358] Peter Thiemann. ML-style typing, lambda lifting, and partial evaluation. In *Proceedings of the 1999 Latin-American Conference on Functional Programming, CLAPF '99*, Recife, Pernambuco, Brazil, March 1999.

[359] Peter Thiemann. Continuation-based partial evaluation without continuations. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003*, number 2694 in Lecture Notes in Computer Science, pages 336–382, San Diego, California, June 2003. Springer-Verlag.

[360] Mads Tofte, editor. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, Amsterdam, The Netherlands, June 1997. ACM Press.

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

[362] Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, 1966.

[363] Philip Wadler. How to replace failure by a list of successes. In Jouannaud [231], pages 113–128.

[364] Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994.

[365] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.

[366] Mitchell Wand. Continuation-based multiprocessing. In Ruth E. Davis and John R. Allen, editors, *Conference Record of the 1980 LISP Conference*, pages 19–28, Stanford, California, August 1980. Reprinted in Higher-Order and Symbolic Computation 12(3):285–299, 1999, with a foreword [372].

[367] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[368] Mitchell Wand. *Induction, Recursion, and Programming*. North-Holland, New York, 1980.

[369] Mitchell Wand. Embedding type structure in semantics. In Deusen and Galil [154], pages 1–6.

[370] Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.

[371] Mitchell Wand. Specifying the correctness of binding-time analysis. *Journal of Functional Programming*, 3(32):365–387, 1993.

[372] Mitchell Wand. Continuation-based multiprocessing revisited. *Higher-Order and Symbolic Computation*, 12(3):283, 1999.

[373] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–38, May 1988. A preliminary version was presented at the 1986 ACM Conference on Lisp and Functional Programming (LFP 1988).

[374] Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In Fisher [178], pages 54–65.

[375] Daniel C. Wang and Andrew W. Appel. Type-safe garbage collectors. In Hanne Riis Nielson, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 166–178, London, United Kingdom, January 2001. ACM Press.

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

[377] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Aiken [10], pages 214–227.

[378] Yong Xiao, Zena M. Ariola, and Michel Mauny. From syntactic theories to interpreters: A specification language and its compilation. In Nachum Dershowitz and Claude Kirchner, editors, *Informal proceedings of the First International Workshop on Rule-Based Programming (RULE 2000)*, Montréal, Canada, September 2000. Available online at `<http://www.loria.fr/~ckirchne/=rule2000/proceedings/>`.

[379] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating proofs of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.

[380] Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

[381] Zhe Yang. Encoding types in ML-like languages. *Theoretical Computer Science*, 315(1):151–190, 2004. A preliminary version was presented at the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP 1998).