

# How / Design Programs

*Jeremy Gibbons*

*BOBKonf, February 2021*

# Continuation-passing style, defunctionalization, and associativity

*Jeremy Gibbons*

*BOBKonf, February 2021*

# 1. Teaser

Binary trees:

**data** *Tree a* = *Tip a* | *Bin (Tree a) (Tree a)*

with flattening to lists:

*flatten*<sub>1</sub> :: *Tree a* → [*a*]

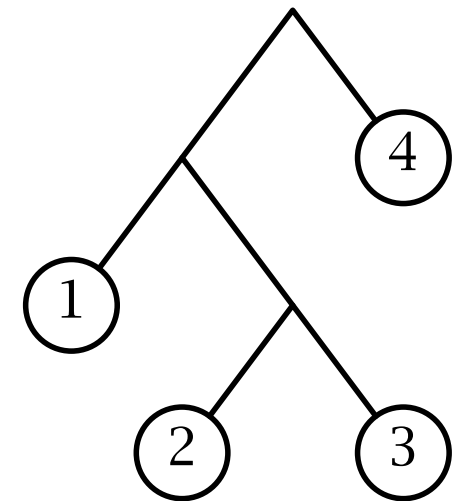
*flatten*<sub>1</sub> (*Tip x*) = [*x*]

*flatten*<sub>1</sub> (*Bin t u*) = *flatten*<sub>1</sub> *t* ++ *flatten*<sub>1</sub> *u*

Takes *quadratic* time, because of left-nested ++s.

How to do it in linear time?

And what does this have to do with abstract machines?



## 2. Factorial

$fact_1 :: Integer \rightarrow Integer$

$fact_1\ 0 = 1$

$fact_1\ n = n \times fact_1\ (n - 1)$

Recursive.

## Continuation-passing style

Introduce *continuation* as accumulating parameter:

$$fact'_2\ n\ k = k\ (fact_1\ n)$$

Then calculate:

$$fact_2 :: Integer \rightarrow Integer$$

$$fact_2\ n = fact'_2\ n\ id$$

$$fact'_2 :: Integer \rightarrow (Integer \rightarrow Integer) \rightarrow Integer$$

$$fact'_2\ 0\ k = k\ 1$$

$$fact'_2\ n\ k = fact'_2\ (n - 1)\ (\lambda m \rightarrow k\ (n \times m))$$

Now *tail-recursive*, but *higher-order*.

## Defunctionalize

The continuations aren't arbitrary  $Integer \rightarrow Integer$  functions:  
always of the form  $id \circ (a \times) \circ (b \times) \circ \dots \circ (c \times)$ .

*Data-refine* this continuation to a list  $[a, b, \dots, c]$ :

$$fact_3 :: Integer \rightarrow Integer$$
$$fact_3\ n = fact'_3\ n\ []$$
$$fact'_3 :: Integer \rightarrow [Integer] \rightarrow Integer$$
$$fact'_3\ 0\ k = product\ k$$
$$fact'_3\ n\ k = fact'_3\ (n - 1)\ (k \# [n])$$

Tail-recursive, *first order*—but uses data structures.

## Associativity

Further data-refine  $[a, b, \dots, c]$  to  $a \times b \times \dots \times c$ .

$fact_4 :: Integer \rightarrow Integer$

$fact_4\ n = fact'_4\ n\ 1$

$fact'_4 :: Integer \rightarrow Integer \rightarrow Integer$

$fact'_4\ 0\ k = k$

$fact'_4\ n\ k = fact'_4\ (n - 1)\ (k \times n)$

Data refinement valid by *associativity*.

Familiar: *tail-recursive*, *first-order*, only *scalar* data.

(This last step wouldn't work for “subfactorial”.)

```
n, k := N, 1;  
{ inv:  $n \geq 0 \wedge k \times n! = N!$  }  
while  $n \neq 0$  do  
     $n, k := n - 1, k \times n$   
end  
{  $k = N!$  }
```

### 3. Reverse

Similarly, naive *quadratic-time* reverse:

$$\text{reverse}_1 :: [a] \rightarrow [a]$$

$$\text{reverse}_1 [] = []$$

$$\text{reverse}_1 (x:xs) = \text{reverse}_1 xs ++ [x]$$

CPS; defunctionalize; associativity:

$$\text{reverse}_3 :: [a] \rightarrow [a]$$

$$\text{reverse}_3 xs = \text{reverse}'_3 xs []$$

$$\text{reverse}'_3 :: [a] \rightarrow [a] \rightarrow [a]$$

$$\text{reverse}'_3 [] k = k$$

$$\text{reverse}'_3 (x:xs) k = \text{reverse}'_3 xs (x:k)$$



## 4. Tree traversal

Binary trees:

**data** *Tree a* = *Tip a* | *Bin (Tree a) (Tree a)*

with flattening to lists:

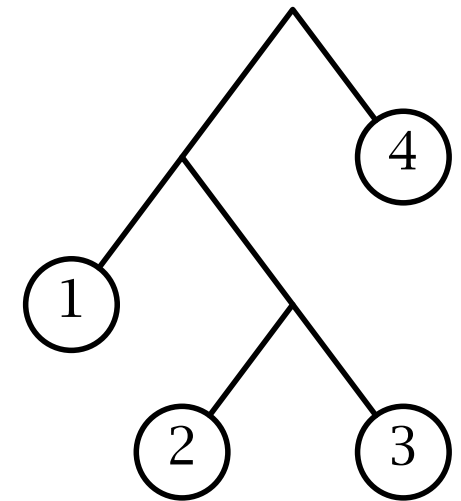
*flatten*<sub>1</sub> :: *Tree a* → [*a*]

*flatten*<sub>1</sub> (*Tip x*) = [*x*]

*flatten*<sub>1</sub> (*Bin t u*) = *flatten*<sub>1</sub> *t* ++ *flatten*<sub>1</sub> *u*

Not tail-recursive;

also *quadratic* time, because of left-nested ++s.



## CPS

Introduce *continuation* as accumulating parameter:

$$\text{flatten}'_2 t k = k (\text{flatten}_1 t)$$

then calculate:

$$\text{flatten}_2 :: \text{Tree } a \rightarrow [a]$$

$$\text{flatten}_2 t = \text{flatten}'_2 t \text{ id}$$

$$\text{flatten}'_2 :: \text{Tree } a \rightarrow ([a] \rightarrow [a]) \rightarrow [a]$$

$$\text{flatten}'_2 (\text{Tip } x) k = k [x]$$

$$\text{flatten}'_2 (\text{Bin } t u) k = \text{flatten}'_2 t (\lambda xs \rightarrow \\ \text{flatten}'_2 u (\lambda ys \rightarrow k (xs ++ ys)))$$

## CPS

Introduce *continuation* as accumulating parameter:

$$\text{flatten}'_2 t k = k (\text{flatten}_1 t)$$

then calculate (NB visit *right child before left*):

$$\text{flatten}_2 :: \text{Tree } a \rightarrow [a]$$

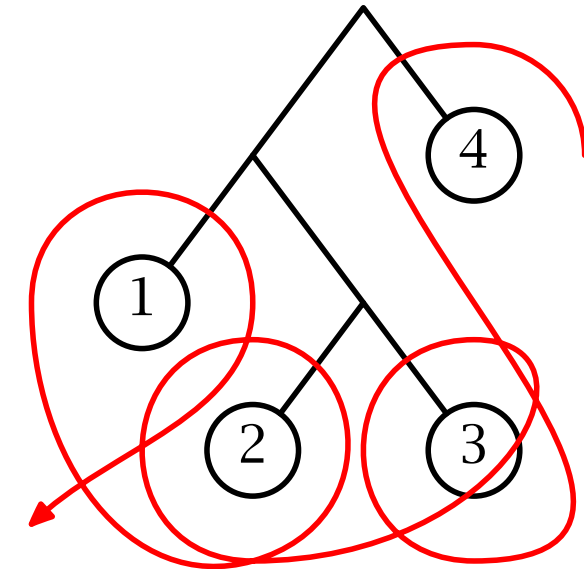
$$\text{flatten}_2 t = \text{flatten}'_2 t \text{ id}$$

$$\text{flatten}'_2 :: \text{Tree } a \rightarrow ([a] \rightarrow [a]) \rightarrow [a]$$

$$\text{flatten}'_2 (\text{Tip } x) k = k [x]$$

$$\text{flatten}'_2 (\text{Bin } t u) k = \text{flatten}'_2 u (\lambda ys \rightarrow \text{flatten}'_2 t (\lambda xs \rightarrow k (xs ++ ys)))$$

Tail-recursive, but higher-order, and still quadratic.



## Defunctionalize

Three ways of *constructing* the continuations  $k$ :

$id$  -- no free variables  
 $\lambda xs \rightarrow k (xs ++ ys)$  -- free variables  $ys :: [a], k$   
 $\lambda ys \rightarrow flatten'_2 t (\lambda xs \rightarrow k (xs ++ ys))$  -- free variables  $t :: Tree\ a, k$

—always a sequence, elements either  $[a]$  or  $Tree\ a$ .

So introduce the following representation:

**type**  $FlattenCont_4\ a = [Either\ [a]\ (Tree\ a)]$

with abstraction function

$flattenabs_4 :: FlattenCont_4\ a \rightarrow ([a] \rightarrow [a])$

## Data refinement

Then data-refine  $flatten_2$  to:

$$flatten_4 :: Tree\ a \rightarrow [a]$$

$$flatten_4\ t = flatten'_4\ t\ []$$

$$flatten'_4 :: Tree\ a \rightarrow FlattenCont_4\ a \rightarrow [a]$$

$$flatten'_4\ (Tip\ x)\ k = flattenabs_4\ k\ [x]$$

$$flatten'_4\ (Bin\ t\ u)\ k = flatten'_4\ u\ (Right\ t : k)$$

$$flattenabs_4\ [] = id$$

$$flattenabs_4\ (Left\ ys : k) = \lambda xs \rightarrow flattenabs_4\ k\ (xs ++ ys)$$

$$flattenabs_4\ (Right\ t : k) = \lambda ys \rightarrow flatten'_4\ t\ (Left\ ys : k)$$

## Example

Let  $xs_i = \text{flatten}_1 t_i$  for  $i = 1, \dots, 4$ .

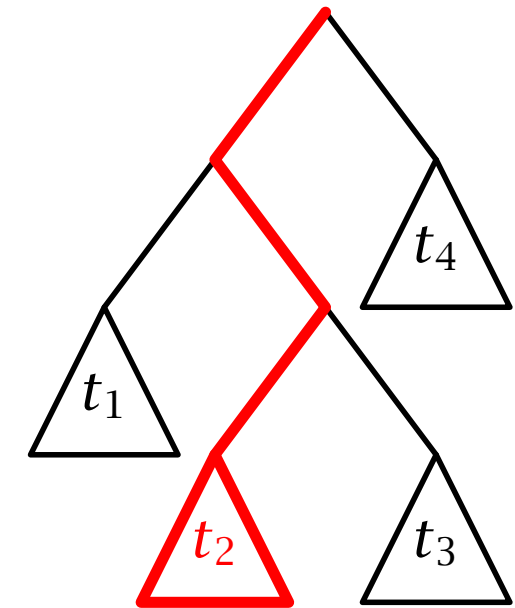
While visiting subtree  $t_2$ , continuation is

$[ \text{Left } xs_3, \text{Right } t_1, \text{Left } xs_4 ]$

Having obtained  $xs_2 = \text{flatten}_1 t_2$ , result is

$(\text{flatten}_1 t_1 ++ (xs_2 ++ xs_3)) ++ xs_4$

By *associativity* of  $++$ , irrelevant relative ordering of *Lefts* (appended) and *Rights* (prepended):

$$\text{flattenabs}_4 k = \text{flattenabs}_4 (\text{map Left (lefts } k) ++ \text{map Right (rights } k))$$


## Associativity

$$\text{flattenabs}_4 k = \text{flattenabs}_4 (\text{Left} (\text{concat} (\text{lefts } k)) \text{ ++ } \text{map Right} (\text{rights } k)))$$

by *associativity* of ++ again, justifying another data refinement:

```

type FlattenCont5 a = ([ a ], [ Tree a ])  -- ‘context’
flatten5 :: Tree a → [ a ]
flatten5 t = flatten'5 t ([ ], [ ])
flatten'5 :: Tree a → FlattenCont5 a → [ a ]
flatten'5 (Tip x) (ys, ts) = flattenabs5 (ys, ts) [ x ]
flatten'5 (Bin t u) (ys, ts) = flatten'5 u (ys, t : ts)
flattenabs5 (ys, [ ]) = λxs → xs ++ ys
flattenabs5 (ys, t : ts) = λys' → flatten'5 t (ys' ++ ys, ts)  -- ys' a singleton

```

## Familiar

Yet another data refinement, combining  
tree-in-focus and stack of trees into just a stack:

$$\text{flatten}_6 :: \text{Tree } a \rightarrow [a]$$
$$\text{flatten}_6 t = \text{flatten}'_6 [t] []$$
$$\text{flatten}'_6 (\text{Tip } x : ts) \quad ys = \text{flatten}'_6 ts (x : ys)$$
$$\text{flatten}'_6 (\text{Bin } t \ u : ts) \quad ys = \text{flatten}'_6 (u : t : ts) \quad ys$$
$$\text{flatten}'_6 [] \quad ys = ys$$

—the *tail-recursive, linear-time* program you might have written.



# Zipper etc

Apply the same process to

$id :: Tree\ a \rightarrow Tree\ a$

yields defunctionalized continuations

$[ Either\ (Tree\ a)\ (Tree\ a) ]$

which is Huet's *zipper*.

J. Functional Programming 7 (5): 549-554, September 1997. Printed in the United Kingdom 549  
© 1997 Cambridge University Press

## FUNCTIONAL PEARL

### *The Zipper*

GÉRARD HUET  
*INRIA Rocquencourt, France*

#### Capsule Review

Almost every programmer has faced the problem of representing a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down the tree. The *Zipper* is Huet's nifty name for a nifty data structure which fulfills this need. I wish I had known of it when I faced this task, because the solution I came up with was not quite so efficient or elegant as the *Zipper*.

#### 1 Introduction

The main drawback to the purely applicative paradigm of programming is that many efficient algorithms use destructive operations in data structures such as bit vectors or character arrays or other mutable hierarchical classification structures, which are not immediately modelled as purely applicative data structures. A well known solution to this problem is called *functional arrays* (Paulson, 1991). For trees, this amounts to modifying an occurrence in a tree non-destructively by copying its *path* from the root of the tree. This is considered tolerable when the data structure is just an object local to some algorithm, the cost being logarithmic compared to the naive solution which copies all the tree. But when the data structure represents some global context, such as the buffer of a text editor, or the database of axioms and lemmas in a proof system, this technique is prohibitive. In this note, we explain a simple solution where tree editing is completely local, the handle on the data not being the original root of the tree, but rather the current position in the tree.

The basic idea is simple: the tree is turned inside-out like a returned glove, pointers from the root to the current position being reversed in a *path structure*. The current *location* holds both the downward current subtree and the upward path. All navigation and modification primitives operate on the location structure. Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name.

The author coined this data-type when designing the core of a structured editor for use as a structure manager for a proof assistant. This simple idea must have been invented on numerous occasions by creative programmers, and the only justification for presenting what ought to be folklore is that it does not appear to have been published, or even to be well-known.

# Zipper etc

Apply the same process to

$$id :: Tree\ a \rightarrow Tree\ a$$

yields defunctionalized continuations

$$[ Either\ (Tree\ a)\ (Tree\ a) ]$$

which is Huet’s zipper. For

$$treeMap :: (a \rightarrow b) \rightarrow Tree\ a \rightarrow Tree\ b$$

yields

$$[ Either\ (Tree\ b)\ (Tree\ a) ]$$

which is McBride’s clowns and jokers.

Almost subtree. The I had ke so effici

The m many e vectors which known this an path le is just the nat some g and let a simpl being t The pointer current navigat and de of clof The for use invente for pre publish

*J. Functional Programming* 7 (5): 549–554, September 1997. Printed in the United Kingdom  
© 1997 Cambridge University Press

549

**Clowns to the Left of me, Jokers to the Right (Pearl)**  
**Dissecting Data Structures**

Conor McBride  
University of Nottingham  
ctm@cs.nott.ac.uk

**Abstract**  
This paper introduces a small but useful generalisation to the ‘derivative’ operation on datatypes underlying Huet’s notion of ‘zipper’ (Huet 1997; McBride 2001; Abbott et al. 2005b), giving a concrete representation to one-hole contexts in data which is undergoing *transformation*. This operator, ‘dissection’, turns a container-like functor into a bifunctor representing a one-hole context in which elements to the left of the hole are distinguished in type from elements to its right.  
I present dissection here as a *generic* program, albeit for polynomial functors only. The notion is certainly applicable more widely, but here I prefer to concentrate on its diverse applications. For a start, map-like operations over the functor and fold-like operations over the recursive data structure it induces can be expressed by tail recursion alone. Further, the derivative is readily recovered from the dissection. Indeed, it is the dissection structure which delivers Huet’s operations for *navigating* zippers.  
The original motivation for dissection was to define ‘division’, capturing the notion of *leftmost* hole, canonically distinguishing values with no elements from those with at least one. Division gives rise to an isomorphism corresponding to the *remainder theorem* in algebra. By way of a larger example, division and dissection are exploited to give a relatively efficient generic algorithm for abstracting all occurrences of one term from another in a first-order syntax.  
The source code for the paper is available online<sup>1</sup> and compiles with recent extensions to the Glasgow Haskell Compiler.  
**Categories and Subject Descriptors** D.1.1 [Programming Techniques]: Applicative (Functional) Programming; I.1.1 [Symbolic and Algebraic Manipulation]: Expressions and Their Representation  
**General Terms** Algorithms, Design, Languages, Theory  
**Keywords** Datatype, Differentiation, Dissection, Division, Generic Programming, Iteration, Polynomial, Stack, Tail Recursion, Traversal, Zipper

<sup>1</sup><http://www.cs.nott.ac.uk/~ctm/GloJo/CJ.1ha>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, without fee, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission under a fee.  
POPL’08, January 7–12, 2008, San Francisco, California, USA.  
Copyright © 2008 ACM 978-1-59593-609-9/08/0001...\$5.00

**1. Introduction**  
There’s an old *Stealer’s Wheel* song with the memorable chorus:  
*‘Clowns to the left of me, jokers to the right,  
Here I am, stuck in the middle with you.’*  
Joe Egan, Gerry Rafferty  
In this paper, I examine what it’s like to be stuck in the middle of traversing and transforming a data structure. I’ll show both you and the Glasgow Haskell Compiler how to calculate the datatype of a ‘freeze-frame’ in a map- or fold-like operation from the datatype being operated on. That is, I’ll explain how to compute a first-class data representation of the control structure underlying map and fold traversals, via an operator which I call *dissection*. Dissection turns out to generalise both the *derivative* operator underlying Huet’s ‘zippers’ (Huet 1997; McBride 2001) and the notion of *division* used to calculate the non-constant part of a polynomial. Let me take you on a journey into the algebra and differential calculus of datatypes, in search of functionality from structure.  
Here’s an example traversal—evaluating a very simple language of expressions:  
**data** Expr = Val Int | Add Expr Expr  
**eval** :: Expr → Int  
**eval** (Val i) = i  
**eval** (Add e1 e2) = **eval** e1 + **eval** e2  
What happens if we freeze a traversal? Typically, we shall have one piece of data ‘in focus’ and a *hole* in the expression where it belongs, with unprocessed data ahead of us and processed data behind. We should expect something a bit like Huet’s ‘zipper’ representation of a one-hole context (Huet 1997), a stack-like structure carrying position information and caching all the out-of-focus data for each node between the hole and the root. However, now we need different sorts of stuff on either side of the hole.  
In the case of our evaluator, suppose we proceed left-to-right. Whenever we face an **Add**, we start by going left into the first operand, recording the second **Expr** to process later; once we have finished with the former, we must go right into the second operand, recording the **Int** returned from the first, as soon as we have both values, we can add them. Correspondingly, a **Stack** of these direction-with-cache choices completely determines where we are in the evaluation process. Let’s make this structure explicit:  
**type** Stack = [Expr → Int]  
Now we can implement an ‘eval machine’—a tail-recursive program, at each stage stuck in the middle with either an expression to decompose, in which case we load the stack and go left, or a value to return, in which case we unload the stack and try to move right.

<sup>2</sup>For brevity, I write + for Either, L for Left and R for Right

Downloaded from <https://www.cambridge.org/core/terms>  
<https://doi.org/10.1017/S1049250X00000001>

287

## 5. Hutton's Razor

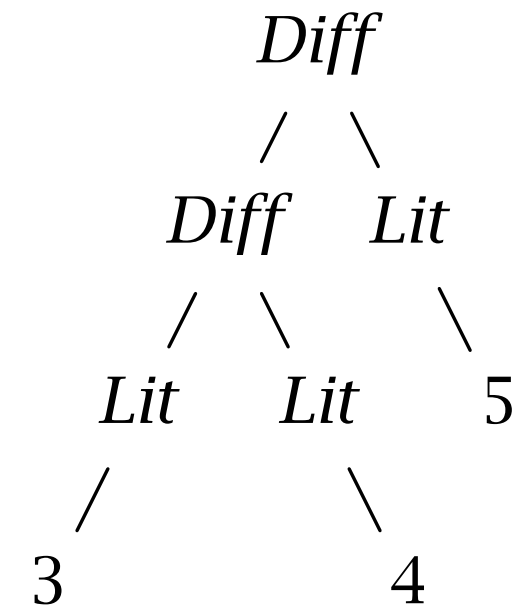
**data** *Expr* = *Lit Integer* | *Diff Expr Expr*

*expr* = *Diff (Diff (Lit 3) (Lit 4)) (Lit 5)*

*eval*<sub>1</sub> :: *Expr* → *Integer*

*eval*<sub>1</sub> (*Lit n*) = *n*

*eval*<sub>1</sub> (*Diff e e'*) = *eval*<sub>1</sub> *e* − *eval*<sub>1</sub> *e'*



## CPS

**type**  $EvalCont_2 = Integer \rightarrow Integer$

$eval_2 :: Expr \rightarrow Integer$

$eval_2\ e = eval'_2\ e\ id$

$eval'_2 :: Expr \rightarrow EvalCont_2 \rightarrow Integer$

$eval'_2\ (Lit\ n)\ k = k\ n$

$eval'_2\ (Diff\ e\ e')\ k = eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow k\ (m - n)))$

Tail-recursive, but higher-order.

## Defunctionalize

**type**  $EvalCont_3 = [ Either\ Expr\ Integer ]$

$eval_3 :: Expr \rightarrow Integer$

$eval_3\ e = eval'_3\ e\ [ ]$

$eval'_3 :: Expr \rightarrow EvalCont_3 \rightarrow Integer$

$eval'_3\ (Lit\ n)\ k = evalabs_3\ k\ n$

$eval'_3\ (Diff\ e\ e')\ k = eval'_3\ e\ (Left\ e' : k)$

$evalabs_3 :: EvalCont_3 \rightarrow (Integer \rightarrow Integer)$

$evalabs_3\ [ ]\ n = n$

$evalabs_3\ (Left\ e' : k)\ m = eval'_3\ e'\ (Right\ m : k)$

$evalabs_3\ (Right\ m : k)\ n = evalabs_3\ k\ (m - n)$

An interpreter, but *not a compiler*: stack contains expressions.

## Where does this compiler come from?

```
data Instr = PushI Integer | SubI
    -- eg [ PushI 3, PushI 4, SubI, PushI 5, SubI ]

compile4 :: Expr → [ Instr ]
compile4 (Lit n)      = [ PushI n ]
compile4 (Diff e e') = compile4 e ++ compile4 e' ++ [ SubI ]

exec4 :: [ Instr ] → [ Integer ] → [ Integer ]
exec4 p s = foldl step s p where
    step ns          (PushI n) = n : ns
    step (n : m : ns) SubI     = (m - n) : ns  -- NB flipping!

eval4 :: Expr → Integer
eval4 e = case exec4 (compile4 e) [ ] of [ n ] → n
```

## ...without pulling rabbits from hats

### *Calculating Correct Compilers*

PATRICK BAHR

Department of Computer Science, University of Copenhagen, Denmark

GRAHAM HUTTON

School of Computer Science, University of Nottingham, UK

---

#### **Abstract**

In this article we present a new approach to the problem of calculating compilers. In particular, we develop a simple but general technique that allows us to derive correct compilers from high-level semantics by systematic calculation, with all details of the implementation of the compilers falling naturally out of the calculation process. Our approach is based upon the use of standard equational reasoning techniques, and has been applied to calculate compilers for a wide range of language features and their combination, including arithmetic expressions, exceptions, state, various forms of lambda calculi, bounded and unbounded loops, non-determinism, and interrupts. All the calculations in the article have been formalised using the Coq proof assistant, which serves as a convenient interactive tool for developing and verifying the calculations.

## ... without pulling rabbits from hats

### *Calculating Correct Compilers*

#### *2.2 Step 2 – Transform into a stack transformer*

The next step is to transform the evaluation function into a version that utilises a stack, in order to make the manipulation of argument values explicit. In particular, rather than returning a single value of type *Int*, we seek to derive a more general evaluation function, *eval<sub>5</sub>*, that takes a stack of integers as an additional argument, and returns a modified stack given by pushing the value of the expression onto the top of the stack. More precisely, if we represent a stack as a list of integers (where the head is the top element)

**type** *Stack* = [*Int*]

then we seek to derive a function

*eval<sub>5</sub>* :: *Expr* → *Stack* → *Stack*

such that:

$$eval_5\ x\ s \quad = \quad eval\ x : s \quad (1)$$

The operator *:* is the list constructor in Haskell, which associates to the right. For example,

In this  
we de  
level s  
falling  
equatio  
langua  
forms  
calcula  
conver



## ... without pulling rabbits from hats

### Calculating Correct Compilers

#### 2.2 Step 2 – Transform into a stack transformer

The next step is to transform the evaluation function into a version that utilises a stack,

in order

return

$eval_S$ ,

given

we rep

**type**

then w

$eval$

such th

The or

#### 2.3 Step 3 – Transform into continuation-passing style

The next step is to transform the new function  $eval_S$  into *continuation-passing style* (CPS)

(Reynolds, 1972), in order to make the flow of control explicit. In particular, we seek to

derive a more general evaluation function,  $eval_C$ , that takes a function from stacks to stacks

(the continuation) as an additional argument, which is used to process the stack that results

from evaluating the expression. More precisely, if we define a type for continuations

**type**  $Cont = Stack \rightarrow Stack$

then we seek to derive a function

$eval_C :: Expr \rightarrow Cont \rightarrow Cont$

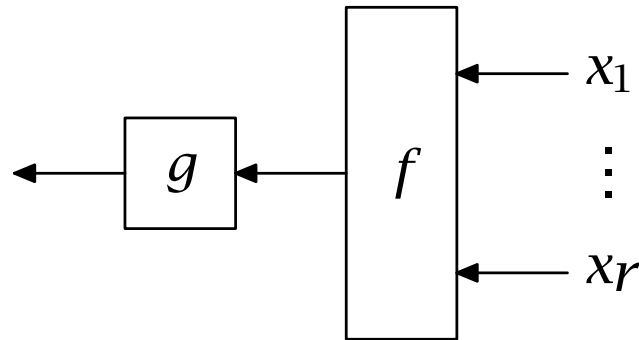
such that:

$$eval_C x c s = c (eval_S x s) \quad (2)$$

## 6. Generalized composition

Generalize composition to propagate *multiple arguments*:

$$b^r g f = \lambda x_1 \dots x_r \rightarrow g (f x_1 \dots x_r)$$



ie

$$b^0 g f = g f$$

$$b^{r+1} g f = \lambda x \rightarrow b^r g f x$$

### Deriving Target Code as a Representation of Continuation Semantics

MITCHELL WAND  
Indiana University

Reynolds' technique for deriving interpreters is extended to derive compilers from continuation semantics. The technique starts by eliminating  $\lambda$ -variables from the semantic equations through the introduction of special-purpose combinators. The semantics of a program phrase may be represented by a term built from these combinators. Then associative and distributive laws are used to simplify the terms. Last, a machine is built to interpret the simplified terms as the functions they represent. The combinators reappear as the instructions of this machine. The technique is illustrated with three examples.

Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—*semantics*; D.3.4 [Programming Languages]: Processors—*code generation; compilers*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*denotational semantics*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*lambda calculus and related systems*

General Terms: Languages, Theory

Additional Key Words and Phrases: Continuations, combinators

#### 1. INTRODUCTION

In this paper, we attack the question of how a denotational semantics for a language is related to an implementation of that language. Typically, one constructs the semantics of a target machine and of a (suitably abstract) compiler and proves a congruence between the two different semantics [12].

Our approach is quite different. Starting with a continuation semantics for the source language, we construct, via a series of transformations and representation decisions, a target machine and a compiler. A typical semantics has functionality

## Implementing generalized composition

Arities:

```
type family Arrow (as :: [ Type ]) (b :: Type) where  
  Arrow '[]      b = b  
  Arrow (a': as) b = a → Arrow as b
```

eg

```
Arrow '[Char, Bool] String = Char → Bool → String
```

Then we can define

```
b :: (b → c) → Arrow as b → Arrow as c
```

(roughly...)

## Installing

Recall:

$$eval_2\ e = eval'_2\ e\ id$$

$$eval'_2\ (Lit\ n) = \lambda k \rightarrow k\ n$$

$$eval'_2\ (Diff\ e\ e') = \lambda k \rightarrow eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow k\ (m - n)))$$

## Installing

Recall:

$$eval_2\ e = eval'_2\ e\ halt$$

$$eval'_2\ (Lit\ n) = ret\ n$$

$$eval'_2\ (Diff\ e\ e') = \lambda k \rightarrow eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow sub\ k\ m\ n))$$

where for later convenience we introduce:

$$halt = id$$

$$ret\ n = \lambda k \rightarrow k\ n$$

$$sub = \lambda k\ m\ n \rightarrow k\ (m - n)$$

## Installing

Recall:

$$eval_2\ e = eval'_2\ e\ halt$$

$$eval'_2\ (Lit\ n) = ret\ n$$

$$eval'_2\ (Diff\ e\ e') = \lambda k \rightarrow eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow sub\ k\ m\ n))$$

Then:

$$\begin{aligned} & eval'_2\ (Diff\ e\ e') \\ = & \quad [[\text{definition}]] \\ & \lambda k \rightarrow eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow sub\ m\ k\ n)) \\ = & \quad [[\text{since } \lambda k \rightarrow g\ (f\ k) \text{ is } b^1\ g\ (\lambda k \rightarrow f\ k)]] \\ & b^1\ (eval'_2\ e)\ (\lambda k\ m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow sub\ k\ m\ n)) \\ = & \quad [[\text{since } \lambda k\ m \rightarrow g\ (f\ k\ m) \text{ is } b^2\ g\ (\lambda k\ m \rightarrow f\ k\ m)]] \\ & b^1\ (eval'_2\ e)\ (b^2\ (eval'_2\ e')\ sub) \end{aligned}$$

## An equivalent evaluator

Rewrite the *Diff* case of  $eval'_2$ :

$eval_5 :: Expr \rightarrow Integer$

$eval_5 e = eval'_5 e \text{ halt}$  **where**

$eval'_5 :: Expr \rightarrow (Integer \rightarrow Integer) \rightarrow Integer$

$eval'_5 (Lit\ n) = ret\ n$

$eval'_5 (Diff\ e\ e') = b^1 (eval'_5\ e) (b^2 (eval'_5\ e')\ sub)$

## Tree-shaped code

```
data ExprRep6 :: [Type] → Type where
  Ret6 :: Integer → ExprRep6 '[]
  Sub6 :: ExprRep6 '[Integer, Integer]
  B61   :: ExprRep6 '[] → ExprRep6 '[Integer] → ExprRep6 '[]
  B62   :: ExprRep6 '[] → ExprRep6 '[Integer, Integer] → ExprRep6 '[Integer]
```

Type index denotes what *extra values* are needed to complete evaluation.



## Representation and interpretation

$rep_6 :: Expr \rightarrow ExprRep_6 \text{ ' [] }$

$rep_6 (Lit\ n) = Ret_6\ n$

$rep_6 (Diff\ e\ e') = B_6^1 (rep_6\ e) (B_6^2 (rep_6\ e')\ Sub_6)$

$abs_6 :: ExprRep_6\ r \rightarrow (Integer \rightarrow Integer) \rightarrow Arrow\ r\ Integer$

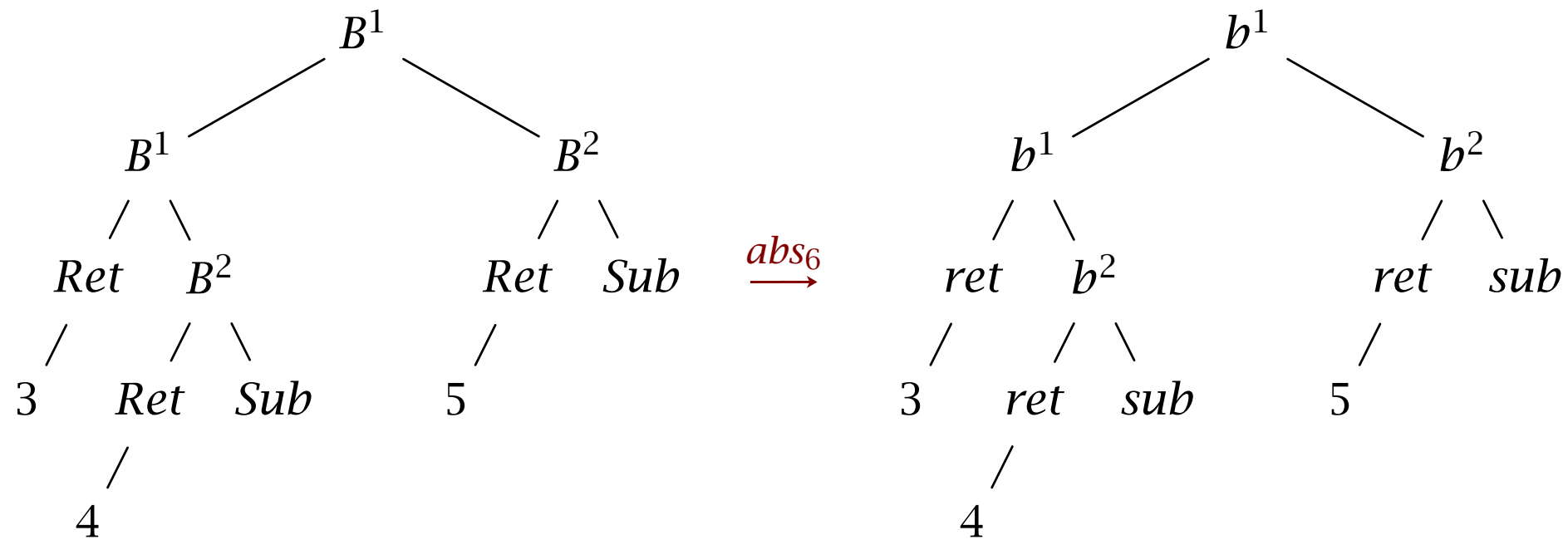
$abs_6 (Ret_6\ n) = ret\ n$

$abs_6\ Sub_6 = sub$

$abs_6 (B_6^1\ x\ y) = b^1 (abs_6\ x) (abs_6\ y)$

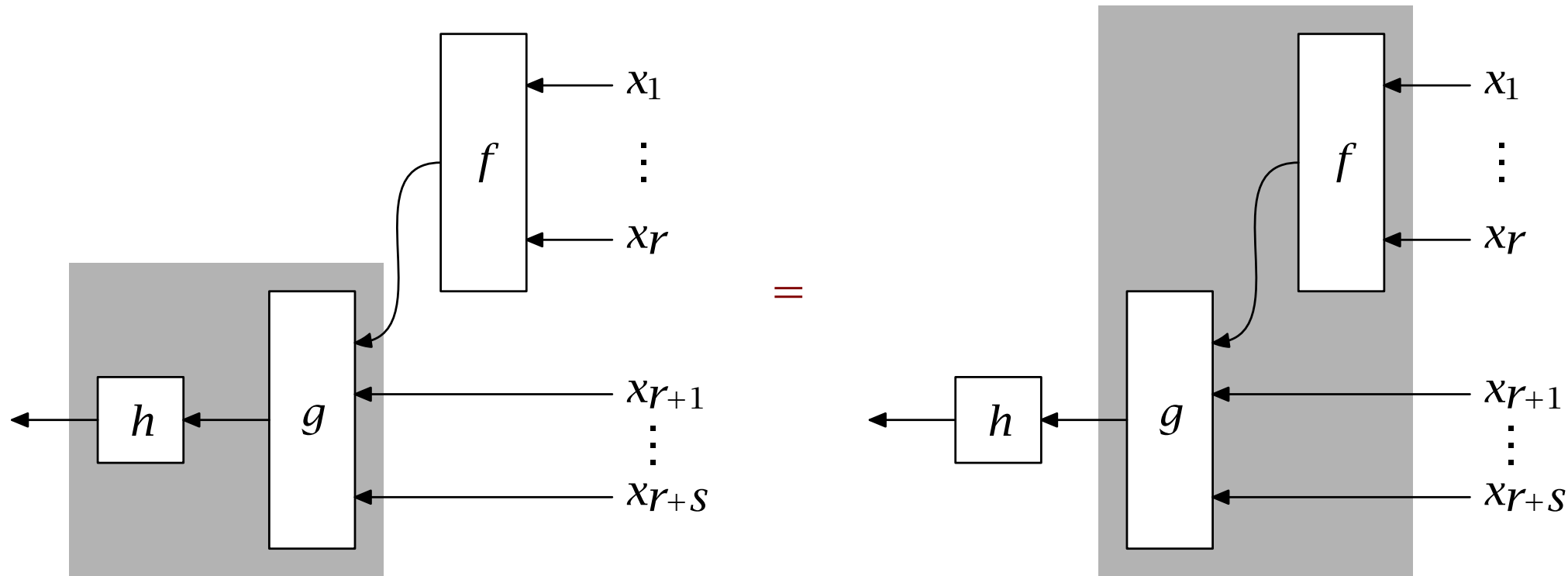
$abs_6 (B_6^2\ x\ y) = b^2 (abs_6\ x) (abs_6\ y)$

## But still tree-shaped



## 7. Associativity

Generalized composition is (of course!) *(pseudo-)associative*:



ie  $b^r (b^{s+1} h g) f = b^{r+s} h (b^r g f)$ . Hence *rotate* tree-shaped code to linear.

## Rotating

$$\begin{aligned}
 & eval_5 \text{ expr} \\
 = & \quad [[ \text{definition of } eval_5, eval'_5; b^0 \text{ is application} \quad ]] \\
 & b^0 (b^1 (b^1 (ret\ 3) (b^2 (ret\ 4) sub)) (b^2 (ret\ 5) sub)) halt \\
 = & \quad [[ \text{pseudo-associativity: } b^0 (b^1 h g) f = b^0 h (b^0 g f) \quad ]] \\
 & b^0 (b^1 (ret\ 3) (b^2 (ret\ 4) sub)) (b^0 (b^2 (ret\ 5) sub) halt) \\
 = & \quad [[ \text{pseudo-associativity: } b^0 (b^1 h g) f = b^0 h (b^0 g f) \quad ]] \\
 & b^0 (ret\ 3) (b^0 (b^2 (ret\ 4) sub) (b^0 (b^2 (ret\ 5) sub) halt)) \\
 = & \quad [[ \text{pseudo-associativity: } b^0 (b^2 h g) f = b^1 h (b^0 g f) \quad ]] \\
 & b^0 (ret\ 3) (b^1 (ret\ 4) (b^0 sub (b^0 (b^2 (ret\ 5) sub) halt))) \\
 = & \quad [[ \text{pseudo-associativity: } b^0 (b^2 h g) f = b^1 h (b^0 g f) \quad ]] \\
 & b^0 (ret\ 3) (b^1 (ret\ 4) (b^0 sub (b^1 (ret\ 5) (b^0 sub halt))))
 \end{aligned}$$

## Linear code

```
data ExprRep7 :: [ Type ] → Type where
  Halt7  ::                               ExprRep7 '[Integer]
  BRet7  :: Integer →
              ExprRep7 (Integer ': r) → ExprRep7 r
  BSub7 :: ExprRep7 (Integer ': r) → ExprRep7 (Integer ': Integer ': r)
```

## Representation and interpretation

$rep_7 :: Expr \rightarrow ExprRep_7 '[]$

$rep_7 (Lit\ n) = BRet_7\ n\ Halt_7$

$rep_7 (Diff\ e\ e') = append\ (rep_7\ e)\ (append\ (rep_7\ e')\ (BSub_7\ Halt_7))$

$abs_7 :: ExprRep_7\ r \rightarrow Arrow\ r\ Integer$

$abs_7\ Halt_7 = halt$

$abs_7 (BRet_7\ n\ k) = ret\ n\ (abs_7\ k)$

$abs_7 (BSub_7\ k) = flip\ (sub\ (abs_7\ k))$

(note flipping subtraction), where...

## Concatenating code

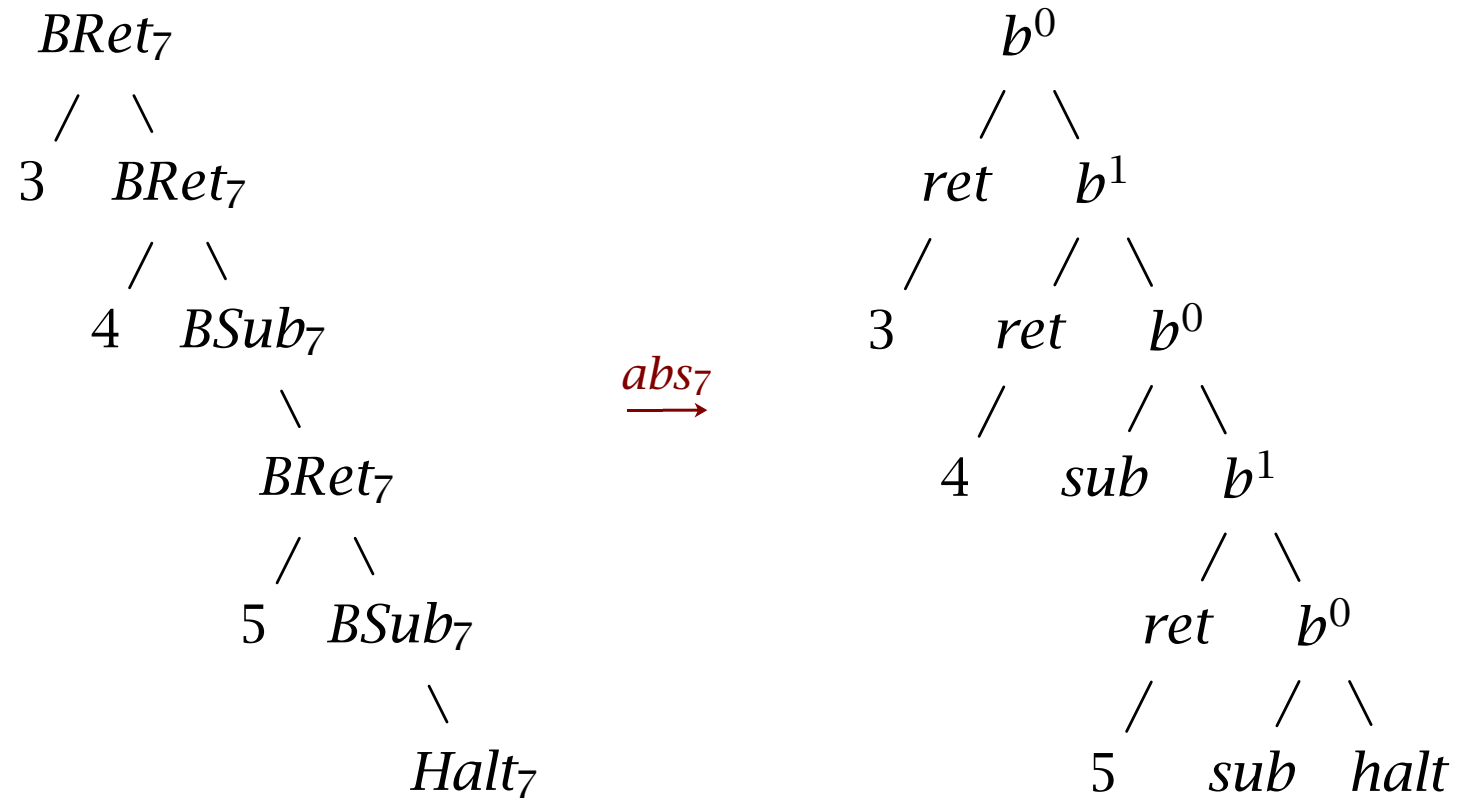
Appending type-level lists:

```
type family Append (as :: [ Type ]) (bs :: [ Type ]) :: [ Type ]  
  where  
    Append '[]      bs = bs  
    Append (a' : as) bs = a' : Append as bs
```

and appending linear programs:

```
append :: ExprRep7 r → ExprRep7 (Integer ' : s) → ExprRep7 (Append r s)  
append Halt7 y      = y  
append (BRet7 n k) y = BRet7 n (append k y)  
append (BSub7 k) y   = BSub7 (append k y)
```

## No longer tree-shaped





## ***This is where the compiler comes from!***

*compile<sub>7</sub> :: Expr → [ Instr ]*

*compile<sub>7</sub> = compileRep<sub>7</sub> ∘ rep<sub>7</sub> where*

*compileRep<sub>7</sub> :: ExprRep<sub>7</sub> r → [ Instr ]*

*compileRep<sub>7</sub> Halt<sub>7</sub> = [ ]*

*compileRep<sub>7</sub> (BRet<sub>7</sub> n k) = PushI n : compileRep<sub>7</sub> k*

*compileRep<sub>7</sub> (BSub<sub>7</sub> k) = SubI : compileRep<sub>7</sub> k*

Indeed:

*compile<sub>7</sub> expr = [ PushI 3, PushI 4, SubI, PushI 5, SubI ]*

## 8. Conclusion

- *accumulating parameters*
- *continuation-passing style* and *defunctionalization*
- Reynolds, Danvy: *recursive* interpreter  $\rightsquigarrow$  *tail-recursive* abstract machine
- many other applications: fast reverse, traversals, zippers...
- but there's usually an appeal to *associativity* there too

### Definitional Interpreters for Higher-Order Programming Languages

John C. Reynolds, Syracuse University

Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters which are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the

#### INTRODUCTION

An important and frequently used method of defining a programming language is to give an interpreter for the language which is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our

### A Functional Correspondence between Evaluators and Abstract Machines

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard  
BRICS<sup>\*</sup>  
Department of Computer Science  
University of Aarhus<sup>†</sup>

#### Abstract

We bridge the gap between functional evaluators and abstract machines for the  $\lambda$ -calculus, using closure conversion, transformation into continuation-passing style, and defunctionalization.

We illustrate this approach by deriving Krivine's abstract machine from an ordinary call-by-name evaluator and by deriving an ordinary call-by-value evaluator from Felleisen et al.'s CEK machine. The first derivation is strikingly simpler than what can be found in the literature. The second one is new. Together, they show that Krivine's abstract machine and the CEK machine correspond to the

#### 1 Introduction and related work

In Hannan and Miller's words [23, Section 7], there are fundamental differences between denotational definitions and definitions of abstract machines. While a functional programmer tends to be familiar with denotational definitions [36], he typically wonders about the following issues:

**Design:** How does one design an abstract machine? How were existing abstract machines, starting with Landin's SECD machine, designed? How does one make variants of an existing abstract machine? How does one extend an existing abstract