How *I* Design Programs

Jeremy Gibbons

BOBKonf, February 2021

Continuation-passing style, defunctionalization, and associativity

Jeremy Gibbons

BOBKonf, February 2021

UNIVERSITY OF OXFORD

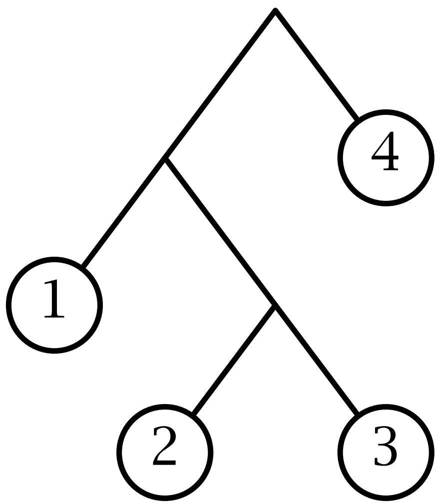
1. Teaser

Binary trees:

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

with flattening to lists:

$$\begin{aligned}
&\mathit{flatten}_1 :: \mathit{Tree}\ a \to [a] \\
&\mathit{flatten}_1\ (\mathit{Tip}\ x) \quad = [x] \\
&\mathit{flatten}_1\ (\mathit{Bin}\ t\ u) = \mathit{flatten}_1\ t + \!\!+ \mathit{flatten}_1\ u
\end{aligned}$$



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

$$\begin{aligned}
&\mathit{fact}_1 :: \mathit{Integer} \rightarrow \mathit{Integer}\\
&\mathit{fact}_1\ 0 = 1\\
&\mathit{fact}_1\ n = n \times \mathit{fact}_1\ (n - 1)
\end{aligned}$$

Recursive.

Continuation-passing style

Introduce *continuation* as accumulating parameter:

$$fact_2'\ n\ k = k\ (fact_1\ n)$$

Then calculate:

$$\begin{aligned}
&fact_2 :: Integer \to Integer \\
&fact_2\ n = fact_2'\ n\ id
\end{aligned}$$

$$\begin{aligned}
&fact_2' :: Integer \to (Integer \to Integer) \to Integer \\
&fact_2'\ 0\ k = k\ 1 \\
&fact_2'\ n\ k = fact_2'\ (n-1)\ (\lambda m \to k\ (n \times m))
\end{aligned}$$

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

Defunctionalize

The continuations aren't arbitrary $\mathit{Integer} \to \mathit{Integer}$ functions: always of the form $\mathit{id} \circ (a\times) \circ (b\times) \circ \cdots \circ (c\times)$.

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

$$
\begin{aligned}
&\mathit{fact}_3 :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fact}_3\ n = \mathit{fact}'_3\ n\ [\,]\\
&\mathit{fact}'_3 :: \mathit{Integer} \to [\,\mathit{Integer}\,] \to \mathit{Integer}\\
&\mathit{fact}'_3\ 0\ k\ = \mathit{product}\ k\\
&\mathit{fact}'_3\ n\ k = \mathit{fact}'_3\ (n-1)\ (k \mathbin{+\!\!+} [\,n\,])
\end{aligned}
$$

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

Associativity

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

$$\begin{array}{l}
fact_4 :: Integer \to Integer \\
fact_4\ n = fact'_4\ n\ 1 \\
fact'_4 :: Integer \to Integer \to Integer \\
fact'_4\ 0\ k = k \\
fact'_4\ n\ k = fact'_4\ (n-1)\ (k \times n)
\end{array}$$

Data refinement valid by *associativity*.

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

(This last step wouldn't work for "subtractorial".)

$$\begin{array}{l}
n, k := N, 1; \\
\{\ \text{inv: } n \geqslant 0 \land k \times n! = N!\ \ \} \\
\mathbf{while}\ n \neq 0\ \mathbf{do} \\
\quad n, k := n-1, k \times n \\
\mathbf{end} \\
\{\ \ k = N!\ \ \}
\end{array}$$

3. Reverse

Similarly, naive *quadratic-time* reverse:

$$
\begin{aligned}
&\mathit{reverse}_1 :: [\,a\,] \to [\,a\,] \\
&\mathit{reverse}_1\ [\,]\qquad\quad = [\,] \\
&\mathit{reverse}_1\ (x : xs) = \mathit{reverse}_1\ xs +\!\!+ [\,x\,]
\end{aligned}
$$

CPS; defunctionalize; associativity:

$$
\begin{aligned}
&\mathit{reverse}_3 :: [\,a\,] \to [\,a\,] \\
&\mathit{reverse}_3\ xs = \mathit{reverse}'_3\ xs\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reverse}'_3 :: [\,a\,] \to [\,a\,] \to [\,a\,] \\
&\mathit{reverse}'_3\ [\,]\qquad\quad k = k \\
&\mathit{reverse}'_3\ (x : xs)\ k = \mathit{reverse}'_3\ xs\ (x : k)
\end{aligned}
$$

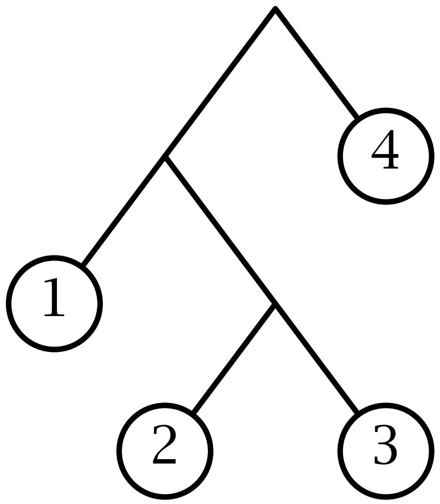
4. Tree traversal

Binary trees:

$$\mathbf{data}\ \mathit{Tree}\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

with flattening to lists:

$$\begin{array}{lcl} \mathit{flatten}_1 :: \mathit{Tree}\ a \to [a] \\ \mathit{flatten}_1\ (\mathit{Tip}\ x) & = & [x] \\ \mathit{flatten}_1\ (\mathit{Bin}\ t\ u) & = & \mathit{flatten}_1\ t +\!\!+ \mathit{flatten}_1\ u \end{array}$$



Not tail-recursive;
also *quadratic* time, because of left-nested $+\!\!+$s.

CPS

Introduce *continuation* as accumulating parameter:

$$flatten_2'\ t\ k = k\ (flatten_1\ t)$$

then calculate:

$$\begin{aligned}
&flatten_2 :: Tree\ a \to [a] \\
&flatten_2\ t = flatten_2'\ t\ id
\end{aligned}$$

$$\begin{aligned}
&flatten_2' :: Tree\ a \to ([a] \to [a]) \to [a] \\
&flatten_2'\ (Tip\ x)\quad k = k\ [x] \\
&flatten_2'\ (Bin\ t\ u)\ k = flatten_2'\ t\ (\lambda xs \to \\
&\qquad\qquad\qquad\qquad flatten_2'\ u\ (\lambda ys \to k\ (xs +\!\!+ ys)))
\end{aligned}$$

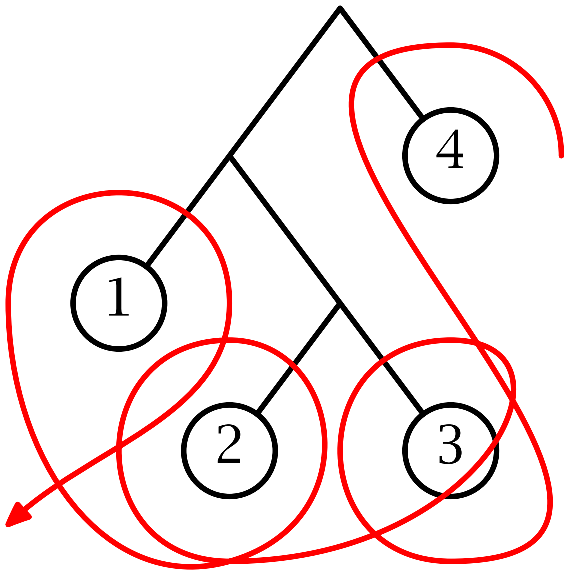
CPS

Introduce *continuation* as accumulating parameter:

$$flatten_2'\ t\ k = k\ (flatten_1\ t)$$

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

$$
\begin{aligned}
&flatten_2 :: Tree\ a \to [\,a\,] \\
&flatten_2\ t = flatten_2'\ t\ id \\
&flatten_2' :: Tree\ a \to ([\,a\,] \to [\,a\,]) \to [\,a\,] \\
&flatten_2'\ (Tip\ x)\quad k = k\ [\,x\,] \\
&flatten_2'\ (Bin\ t\ u)\ k = flatten_2'\ u\ (\lambda ys \to \\
&\qquad\qquad flatten_2'\ t\ (\lambda xs \to k\ (xs + ys)))
\end{aligned}
$$



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

Defunctionalize

Three ways of *constructing* the continuations k:

$$\begin{array}{ll} id & \text{-- no free variables} \\ \lambda xs \rightarrow k\ (xs \mathbin{+\!\!+} ys) & \text{-- free variables } ys :: [\,a\,], k \\ \lambda ys \rightarrow flatten_2'\ t\ (\lambda xs \rightarrow k\ (xs \mathbin{+\!\!+} ys)) & \text{-- free variables } t :: Tree\ a, k \end{array}$$

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

So introduce the following representation:

$$\mathbf{type}\ FlattenCont_4\ a = [\,Either\ [\,a\,]\ (Tree\ a)\,]$$

with abstraction function

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

Data refinement

Then data-refine $\mathit{flatten}_2$ to:

$$
\begin{aligned}
&\mathit{flatten}_4 :: \mathit{Tree}\ a \to [\,a\,] \\
&\mathit{flatten}_4\ t = \mathit{flatten}'_4\ t\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{flatten}'_4 :: \mathit{Tree}\ a \to \mathit{FlattenCont}_4\ a \to [\,a\,] \\
&\mathit{flatten}'_4\ (\mathit{Tip}\ x)\ k \quad = \mathit{flattenabs}_4\ k\ [\,x\,] \\
&\mathit{flatten}'_4\ (\mathit{Bin}\ t\ u)\ k = \mathit{flatten}'_4\ u\ (\mathit{Right}\ t : k)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flattenabs}_4\ [\,] \qquad\qquad &= \mathit{id} \\
\mathit{flattenabs}_4\ (\mathit{Left}\ \mathit{ys} : k) &= \lambda \mathit{xs} \to \mathit{flattenabs}_4\ k\ (\mathit{xs} +\!\!+\ \mathit{ys}) \\
\mathit{flattenabs}_4\ (\mathit{Right}\ t : k) &= \lambda \mathit{ys} \to \mathit{flatten}'_4\ t\ (\mathit{Left}\ \mathit{ys} : k)
\end{aligned}
$$

Example

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

While visiting subtree t_2, continuation is

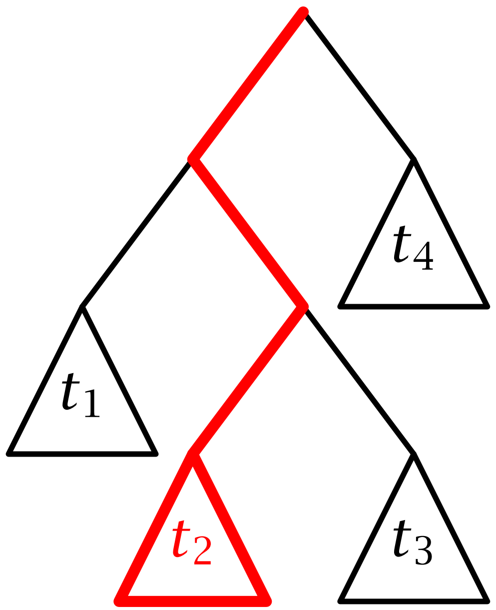
$$[\,\mathit{Left}\ xs_3, \mathit{Right}\ t_1, \mathit{Left}\ xs_4\,]$$

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

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

By *associativity* of $+\!\!+$, irrelevant relative ordering of *Left*s (appended) and *Right*s (prepended):

$$\begin{aligned}\mathit{flattenabs}_4\ k = \mathit{flattenabs}_4\ (&\mathit{map}\ \mathit{Left}\ (\mathit{lefts}\ k) +\!\!+ \\ &\mathit{map}\ \mathit{Right}\ (\mathit{rights}\ k))\end{aligned}$$



Associativity

$$\begin{aligned} \mathit{flattenabs}_4\ k = \mathit{flattenabs}_4\ (\mathit{Left}\ (\mathit{concat}\ (\mathit{lefts}\ k))\ +\!\!+ \\ \mathit{map}\ \mathit{Right}\ (\mathit{rights}\ k))) \end{aligned}$$

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

$$\textbf{type}\ \mathit{FlattenCont}_5\ a = ([a], [\mathit{Tree}\ a]) \quad \text{-- 'context'}$$

$$\begin{aligned} &\mathit{flatten}_5 :: \mathit{Tree}\ a \to [a] \\ &\mathit{flatten}_5\ t = \mathit{flatten}'_5\ t\ ([\,], [\,]) \end{aligned}$$

$$\begin{aligned} &\mathit{flatten}'_5 :: \mathit{Tree}\ a \to \mathit{FlattenCont}_5\ a \to [a] \\ &\mathit{flatten}'_5\ (\mathit{Tip}\ x)\quad (ys, ts) = \mathit{flattenabs}_5\ (ys, ts)\ [x] \\ &\mathit{flatten}'_5\ (\mathit{Bin}\ t\ u)\ (ys, ts) = \mathit{flatten}'_5\ u\ (ys, t : ts) \end{aligned}$$

$$\begin{aligned} &\mathit{flattenabs}_5\ (ys, [\,]) \quad\ \ = \lambda xs \to xs +\!\!+ ys \\ &\mathit{flattenabs}_5\ (ys, t : ts) = \lambda ys' \to \mathit{flatten}'_5\ t\ (ys' +\!\!+ ys, ts) \quad \text{-- } ys' \text{ a singleton} \end{aligned}$$

Familiar

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

$$
\begin{aligned}
&\mathit{flatten}_6 :: \mathit{Tree}\ a \to [a] \\
&\mathit{flatten}_6\ t = \mathit{flatten}'_6\ [t]\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{flatten}'_6\ (\mathit{Tip}\ x : ts)\quad ys &= \mathit{flatten}'_6\ ts\ (x : ys) \\
\mathit{flatten}'_6\ (\mathit{Bin}\ t\ u : ts)\ ys &= \mathit{flatten}'_6\ (u : t : ts)\ ys \\
\mathit{flatten}'_6\ [\,]\ ys &= ys
\end{aligned}
$$

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

Zipper etc

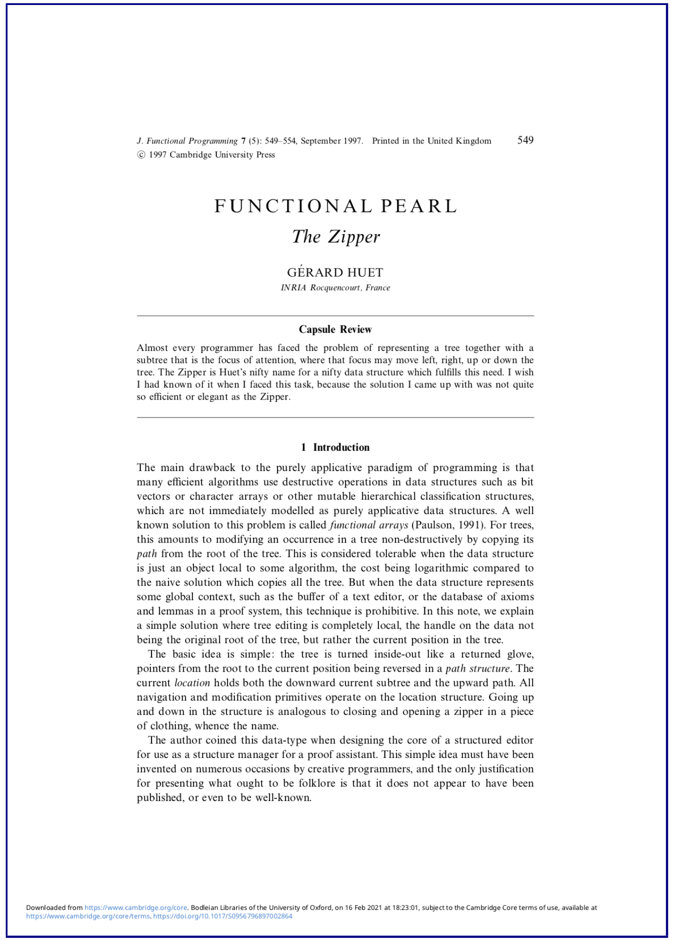
Apply the same process to

$$id :: Tree\ a \to Tree\ a$$

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

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

$$\mathit{id} :: \mathit{Tree}\ a \to \mathit{Tree}\ a$$

yields defunctionalized continuations

$$[\,\mathit{Either}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\,]$$

which is Huet's *zipper*. For

$$\mathit{treeMap} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b$$

yields

$$[\,\mathit{Either}\ (\mathit{Tree}\ b)\ (\mathit{Tree}\ a)\,]$$

which is McBride's *clowns and jokers*.


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

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

1. Introduction

287

5. Hutton's Razor

$$\mathbf{data}\ Expr = Lit\ Integer \mid Diff\ Expr\ Expr$$
$$expr = Diff\ (Diff\ (Lit\ 3)\ (Lit\ 4))\ (Lit\ 5)$$

$$\begin{aligned}
&eval_1 :: Expr \to Integer\\
&eval_1\ (Lit\ n) \quad\quad\ = n\\
&eval_1\ (Diff\ e\ e') = eval_1\ e - eval_1\ e'
\end{aligned}$$

```
        Diff
        /  \
     Diff   Lit
     /  \     \
   Lit  Lit    5
   /      \
  3        4
```

CPS

$$\textbf{type}\ EvalCont_2 = Integer \rightarrow Integer$$

$$\begin{aligned}
&eval_2 :: Expr \rightarrow Integer\\
&eval_2\ e = eval'_2\ e\ id
\end{aligned}$$

$$\begin{aligned}
&eval'_2 :: Expr \rightarrow EvalCont_2 \rightarrow Integer\\
&eval'_2\ (Lit\ n)\qquad k = k\ n\\
&eval'_2\ (Diff\ e\ e')\ k = eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow k\ (m - n)))
\end{aligned}$$

Tail-recursive, but higher-order.

Defunctionalize

$$\textbf{type}\ \mathit{EvalCont}_3 = [\mathit{Either\ Expr\ Integer}]$$

$$\begin{aligned}
&\mathit{eval}_3 :: \mathit{Expr} \rightarrow \mathit{Integer}\\
&\mathit{eval}_3\ e = \mathit{eval}'_3\ e\ [\,]
\end{aligned}$$

$$\begin{aligned}
&\mathit{eval}'_3 :: \mathit{Expr} \rightarrow \mathit{EvalCont}_3 \rightarrow \mathit{Integer}\\
&\mathit{eval}'_3\ (\mathit{Lit}\ n)\qquad\ k = \mathit{evalabs}_3\ k\ n\\
&\mathit{eval}'_3\ (\mathit{Diff}\ e\ e')\ k = \mathit{eval}'_3\ e\ (\mathit{Left}\ e' : k)
\end{aligned}$$

$$\begin{aligned}
&\mathit{evalabs}_3 :: \mathit{EvalCont}_3 \rightarrow (\mathit{Integer} \rightarrow \mathit{Integer})\\
&\mathit{evalabs}_3\ [\,]\qquad\qquad\qquad n\ = n\\
&\mathit{evalabs}_3\ (\mathit{Left}\ e' : k)\quad\ m = \mathit{eval}'_3\ e'\ (\mathit{Right}\ m : k)\\
&\mathit{evalabs}_3\ (\mathit{Right}\ m : k)\ n\ = \mathit{evalabs}_3\ k\ (m - n)
\end{aligned}$$

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

Where does this compiler come from?

$$
\begin{array}{l}
\textbf{data}\ \mathit{Instr} = \mathit{PushI}\ \mathit{Integer} \mid \mathit{SubI} \\
\qquad\qquad \text{-- eg}\ [\mathit{PushI}\ 3, \mathit{PushI}\ 4, \mathit{SubI}, \mathit{PushI}\ 5, \mathit{SubI}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{compile}_4 :: \mathit{Expr} \to [\mathit{Instr}] \\
\mathit{compile}_4\ (\mathit{Lit}\ n) \qquad\ \ = [\mathit{PushI}\ n] \\
\mathit{compile}_4\ (\mathit{Diff}\ e\ e') = \mathit{compile}_4\ e +\!\!+\ \mathit{compile}_4\ e' +\!\!+ [\mathit{SubI}]
\end{array}
$$

$$
\begin{array}{l}
\mathit{exec}_4 :: [\mathit{Instr}] \to [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{exec}_4\ p\ s = \mathit{foldl}\ \mathit{step}\ s\ p\ \ \textbf{where} \\
\quad \mathit{step}\ \mathit{ns} \qquad\qquad\quad (\mathit{PushI}\ n) = n : \mathit{ns} \\
\quad \mathit{step}\ (n : m : \mathit{ns})\ \mathit{SubI} \qquad\ = (m - n) : \mathit{ns} \quad \text{-- NB flipping!}
\end{array}
$$

$$
\begin{array}{l}
\mathit{eval}_4 :: \mathit{Expr} \to \mathit{Integer} \\
\mathit{eval}_4\ e = \textbf{case}\ \mathit{exec}_4\ (\mathit{compile}_4\ e)\ [\,]\ \textbf{of}\ [n] \to n
\end{array}
$$

. . . 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, $eval_S$, 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)

$$\textbf{type}\ Stack = [Int]$$

then we seek to derive a function

$$eval_S :: Expr \to Stack \to Stack$$

such that:

$$eval_S\ x\ s \quad = \quad eval\ x : s \qquad (1)$$

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

. . . 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 ord…

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

$$\textbf{type}\ Cont = Stack \rightarrow Stack$$

then we seek to derive a function

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

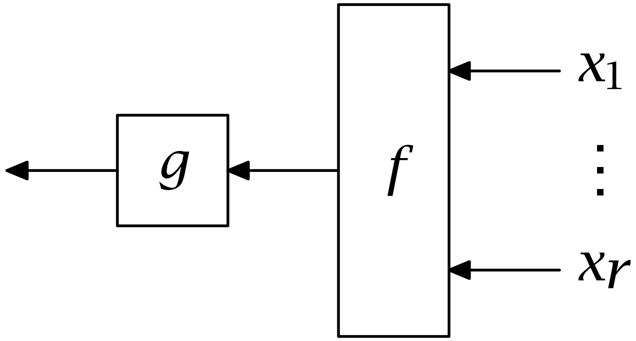
such that:

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

6. Generalized composition

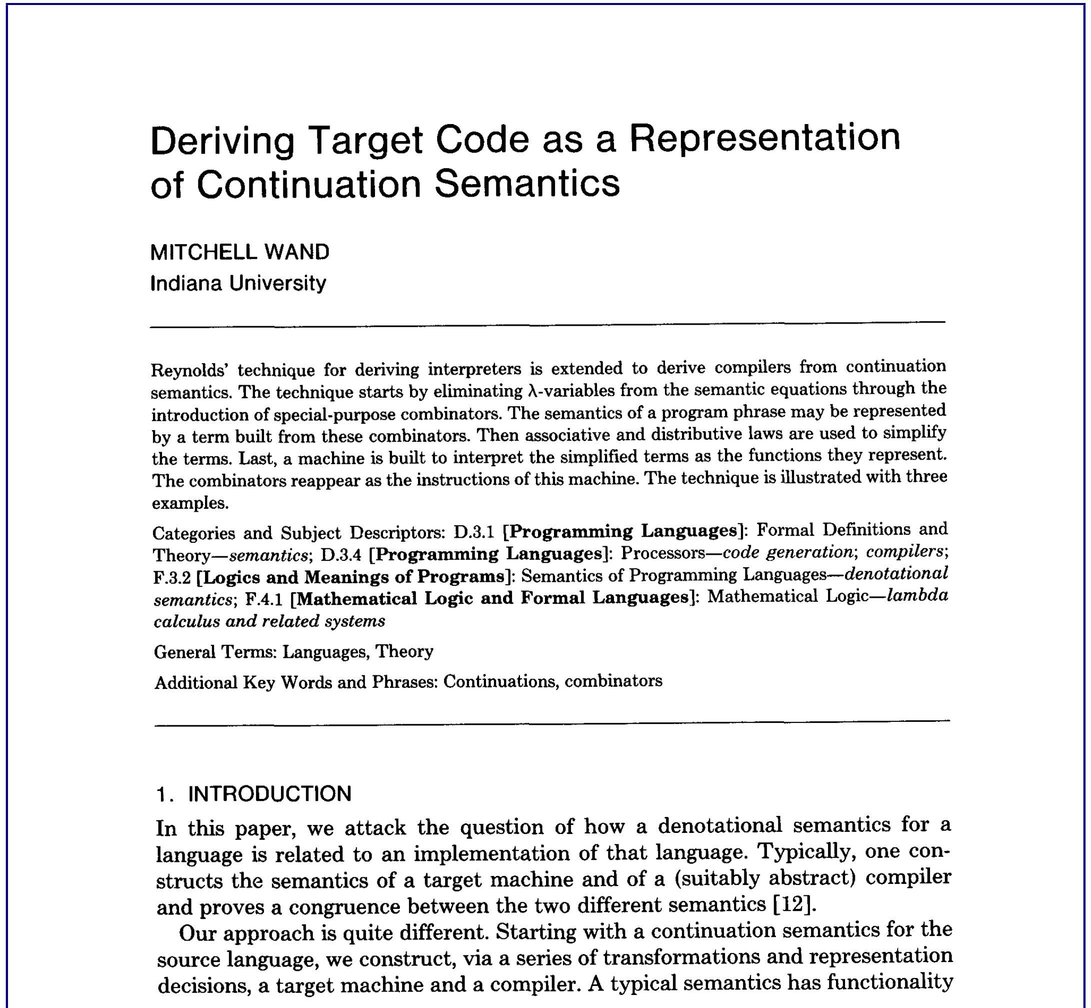
Generalize composition to propagate *multiple arguments*:

$$b^r\ g\ f = \lambda x_1 \ldots x_r \to g\ (f\ x_1 \ldots x_r)$$



ie

$$\begin{aligned} b^0\ g\ f &= g\ f \\ b^{r+1}\ g\ f &= \lambda x \to b^r\ g\ f\ x \end{aligned}$$



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 λ-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:

$$
\begin{array}{l}
\textbf{type family}\ \mathit{Arrow}\ (\mathit{as} :: [\,\mathit{Type}\,])\ (b :: \mathit{Type})\ \ \textbf{where} \\
\quad \mathit{Arrow}\ '[\,]\qquad\quad b = b \\
\quad \mathit{Arrow}\ (a\ ':\mathit{as})\ b = a \to \mathit{Arrow}\ \mathit{as}\ b
\end{array}
$$

eg

$$\mathit{Arrow}\ '[\mathit{Char}, \mathit{Bool}]\ \mathit{String} = \mathit{Char} \to \mathit{Bool} \to \mathit{String}$$

Then we can define

$$b :: (b \to c) \to \mathit{Arrow}\ \mathit{as}\ b \to \mathit{Arrow}\ \mathit{as}\ c$$

(roughly. . .)

Installing

Recall:

$$
\begin{aligned}
&\mathit{eval}_2\ e = \mathit{eval}_2'\ e\ \mathit{id}\\
&\mathit{eval}_2'\ (\mathit{Lit}\ n) \qquad\ \ = \lambda k \to k\ n\\
&\mathit{eval}_2'\ (\mathit{Diff}\ e\ e') = \lambda k \to \mathit{eval}_2'\ e\ (\lambda m \to \mathit{eval}_2'\ e'\ (\lambda n \to k\ (m - n)))
\end{aligned}
$$

Installing

Recall:

$$
\begin{aligned}
&eval_2\ e = eval_2'\ e\ halt\\
&eval_2'\ (Lit\ n) \qquad\quad = ret\ n\\
&eval_2'\ (Diff\ e\ e') = \lambda k \to eval_2'\ e\ (\lambda m \to eval_2'\ e'\ (\lambda n \to sub\ k\ m\ n))
\end{aligned}
$$

where for later convenience we introduce:

$$
\begin{aligned}
halt\ \ &= id\\
ret\ n &= \lambda k \to k\ n\\
sub\ \ &= \lambda k\ m\ n \to k\ (m - n)
\end{aligned}
$$

Installing

Recall:

$$eval_2\ e = eval'_2\ e\ halt$$
$$eval'_2\ (Lit\ n) \quad = ret\ n$$
$$eval'_2\ (Diff\ e\ e') = \lambda k \rightarrow eval'_2\ e\ (\lambda m \rightarrow eval'_2\ e'\ (\lambda n \rightarrow sub\ k\ m\ n))$$

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 $eval_2'$:

$$
\begin{aligned}
&eval_5 :: Expr \to Integer \\
&eval_5\ e = eval_5'\ e\ halt\ \ \mathbf{where} \\
&\quad eval_5' :: Expr \to (Integer \to Integer) \to Integer \\
&\quad eval_5'\ (Lit\ n) \qquad = ret\ n \\
&\quad eval_5'\ (Diff\ e\ e') = b^1\ (eval_5'\ e)\ (b^2\ (eval_5'\ e')\ sub)
\end{aligned}
$$

Tree-shaped code

$$
\begin{array}{lll}
\textbf{data}\ \mathit{ExprRep}_6 :: [\,\mathit{Type}\,] \to \mathit{Type}\ \ \textbf{where} & & \\
\quad \mathit{Ret}_6 & :: \mathit{Integer} \to & \mathit{ExprRep}_6\ '[\,] \\
\quad \mathit{Sub}_6 & :: & \mathit{ExprRep}_6\ '[\mathit{Integer}, \mathit{Integer}] \\
\quad B^1_6 & :: \mathit{ExprRep}_6\ '[\,] \to & \\
 & \quad \mathit{ExprRep}_6\ '[\mathit{Integer}] \to & \mathit{ExprRep}_6\ '[\,] \\
\quad B^2_6 & :: \mathit{ExprRep}_6\ '[\,] \to & \\
 & \quad \mathit{ExprRep}_6\ '[\mathit{Integer}, \mathit{Integer}] \to & \mathit{ExprRep}_6\ '[\mathit{Integer}]
\end{array}
$$

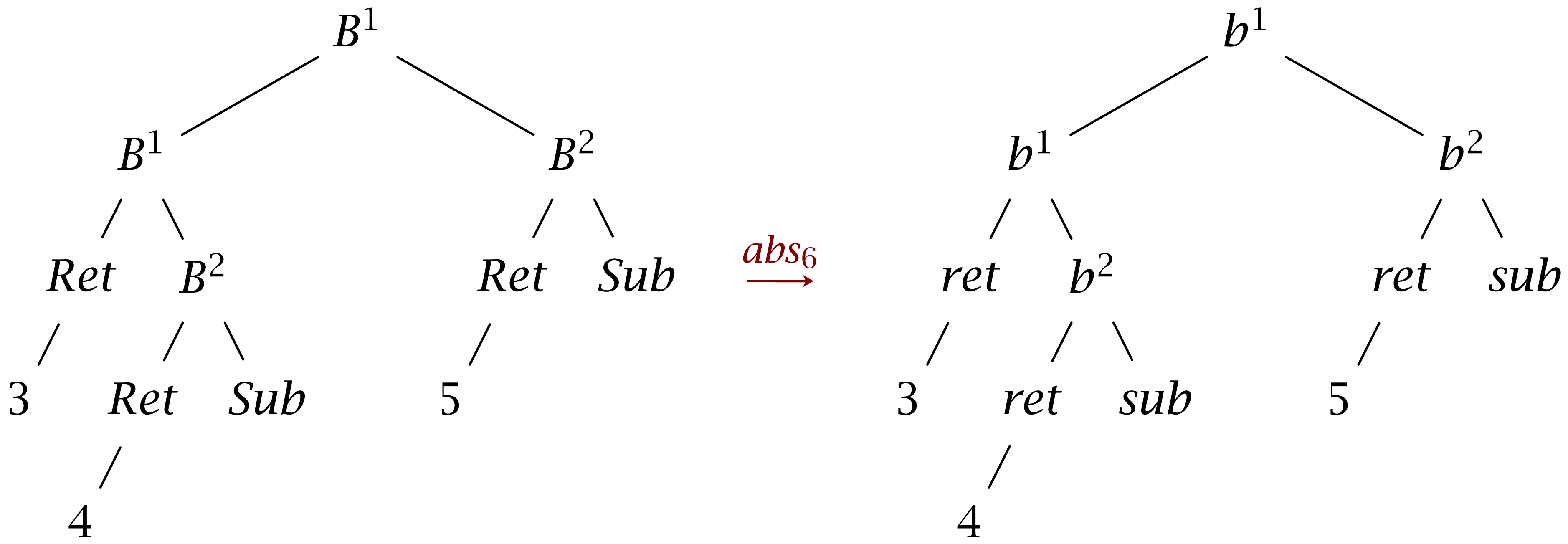
Type index denotes what *extra values* are needed to complete evaluation.

Representation and interpretation

$$
\begin{aligned}
&rep_6 :: Expr \rightarrow ExprRep_6\ '[\,]\\
&rep_6\ (Lit\ n) \qquad\quad = Ret_6\ n\\
&rep_6\ (Diff\ e\ e') = B_6^1\ (rep_6\ e)\ (B_6^2\ (rep_6\ e')\ Sub_6)
\end{aligned}
$$

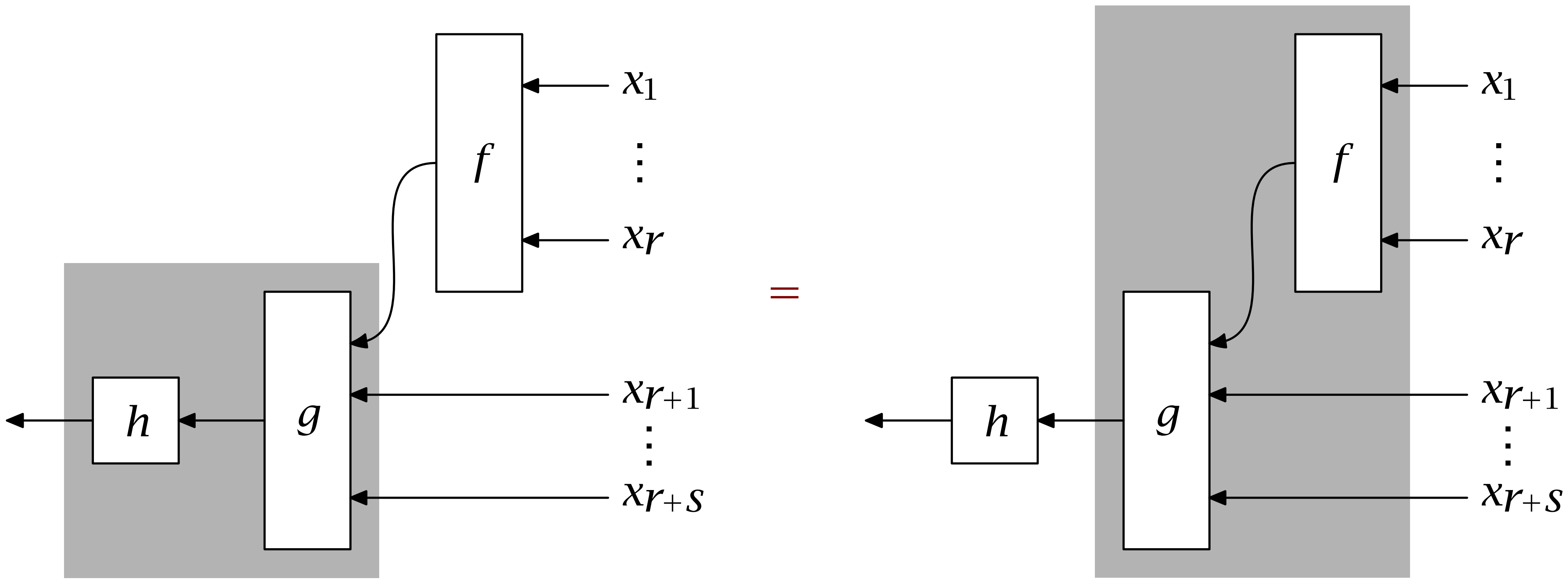
$$
\begin{aligned}
&abs_6 :: ExprRep_6\ r \rightarrow (Integer \rightarrow Integer) \rightarrow Arrow\ r\ Integer\\
&abs_6\ (Ret_6\ n) = ret\ n\\
&abs_6\ Sub_6 \qquad\ = sub\\
&abs_6\ (B_6^1\ x\ y)\ = b^1\ (abs_6\ x)\ (abs_6\ y)\\
&abs_6\ (B_6^2\ x\ y)\ = b^2\ (abs_6\ x)\ (abs_6\ y)
\end{aligned}
$$

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

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

Linear code

$$
\begin{array}{l}
\mathbf{data}\ \mathit{ExprRep}_7 :: [\mathit{Type}] \to \mathit{Type}\ \ \mathbf{where} \\
\quad \mathit{Halt}_7\ \ :: \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ExprRep}_7\ '[\mathit{Integer}] \\
\quad \mathit{BRet}_7\ :: \mathit{Integer} \to \\
\qquad\qquad\quad \mathit{ExprRep}_7\ (\mathit{Integer}\ ': r) \to \mathit{ExprRep}_7\ r \\
\quad \mathit{BSub}_7 :: \mathit{ExprRep}_7\ (\mathit{Integer}\ ': r) \to \mathit{ExprRep}_7\ (\mathit{Integer}\ ': \mathit{Integer}\ ': r)
\end{array}
$$

Representation and interpretation

$$
\begin{aligned}
&\mathit{rep}_7 :: \mathit{Expr} \to \mathit{ExprRep}_7\ '[\,]\\
&\mathit{rep}_7\ (\mathit{Lit}\ n) && = \mathit{BRet}_7\ n\ \mathit{Halt}_7\\
&\mathit{rep}_7\ (\mathit{Diff}\ e\ e') && = \mathit{append}\ (\mathit{rep}_7\ e)\ (\mathit{append}\ (\mathit{rep}_7\ e')\ (\mathit{BSub}_7\ \mathit{Halt}_7))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{abs}_7 :: \mathit{ExprRep}_7\ r \to \mathit{Arrow}\ r\ \mathit{Integer}\\
&\mathit{abs}_7\ \mathit{Halt}_7 && = \mathit{halt}\\
&\mathit{abs}_7\ (\mathit{BRet}_7\ n\ k) && = \mathit{ret}\ n\ (\mathit{abs}_7\ k)\\
&\mathit{abs}_7\ (\mathit{BSub}_7\ k) && = \mathit{flip}\ (\mathit{sub}\ (\mathit{abs}_7\ k))
\end{aligned}
$$

(note flipping subtraction), where…

Concatenating code

Appending type-level lists:

$$
\begin{array}{l}
\textbf{type family } \mathit{Append}\ (\mathit{as} :: [\,\mathit{Type}\,])\ (\mathit{bs} :: [\,\mathit{Type}\,]) :: [\,\mathit{Type}\,] \\
\quad \textbf{where} \\
\qquad \mathit{Append}\ '[\,]\ \qquad\quad \mathit{bs} = \mathit{bs} \\
\qquad \mathit{Append}\ (a\ ': \mathit{as})\ \mathit{bs} = a\ ': \mathit{Append}\ \mathit{as}\ \mathit{bs}
\end{array}
$$

and appending linear programs:

$$
\begin{array}{l}
\mathit{append} :: \mathit{ExprRep}_7\ r \to \mathit{ExprRep}_7\ (\mathit{Integer}\ ': s) \to \mathit{ExprRep}_7\ (\mathit{Append}\ r\ s) \\
\mathit{append}\ \mathit{Halt}_7\ y \qquad\qquad\ = y \\
\mathit{append}\ (\mathit{BRet}_7\ n\ k)\ y = \mathit{BRet}_7\ n\ (\mathit{append}\ k\ y) \\
\mathit{append}\ (\mathit{BSub}_7\ k)\ y \quad\ = \mathit{BSub}_7\ (\mathit{append}\ k\ y)
\end{array}
$$

No longer tree-shaped

```
  BRet_7                                       b^0
   /  \                                       /   \
  3   BRet_7                               ret     b^1
       /  \                                /       /   \
      4   BSub_7          abs_7           3     ret     b^0
             \           ------>                /       /   \
            BRet_7                             4     sub     b^1
             /  \                                            /   \
            5   BSub_7                                    ret     b^0
                   \                                      /       /   \
                   Halt_7                                5     sub    halt
```

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

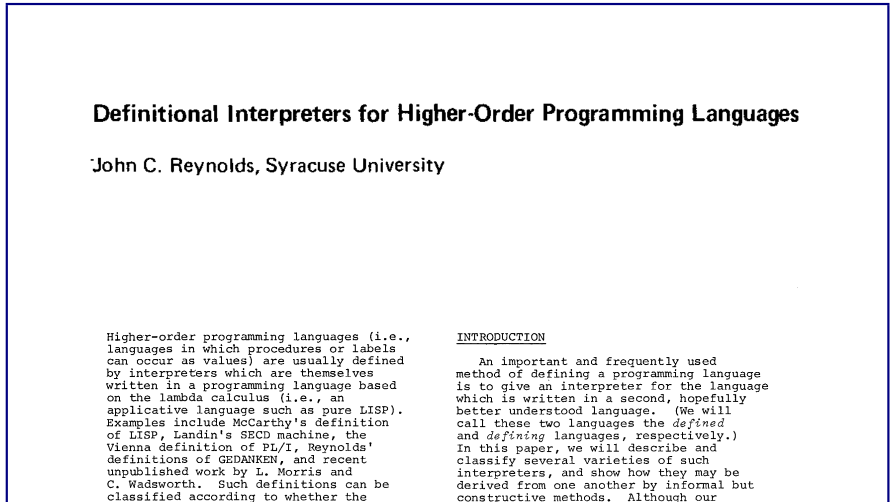
$$
\begin{aligned}
&\mathit{compile}_7 :: \mathit{Expr} \to [\,\mathit{Instr}\,] \\
&\mathit{compile}_7 = \mathit{compileRep}_7 \circ \mathit{rep}_7 \ \ \textbf{where} \\
&\quad \mathit{compileRep}_7 :: \mathit{ExprRep}_7\ r \to [\,\mathit{Instr}\,] \\
&\quad \mathit{compileRep}_7\ \mathit{Halt}_7 \qquad\qquad = [\,] \\
&\quad \mathit{compileRep}_7\ (\mathit{BRet}_7\ n\ k) = \mathit{PushI}\ n : \mathit{compileRep}_7\ k \\
&\quad \mathit{compileRep}_7\ (\mathit{BSub}_7\ k) \quad\ = \mathit{SubI} : \mathit{compileRep}_7\ k
\end{aligned}
$$

Indeed:

$$\mathit{compile}_7\ \mathit{expr} = [\,\mathit{PushI}\ 3, \mathit{PushI}\ 4, \mathit{SubI}, \mathit{PushI}\ 5, \mathit{SubI}\,]$$

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

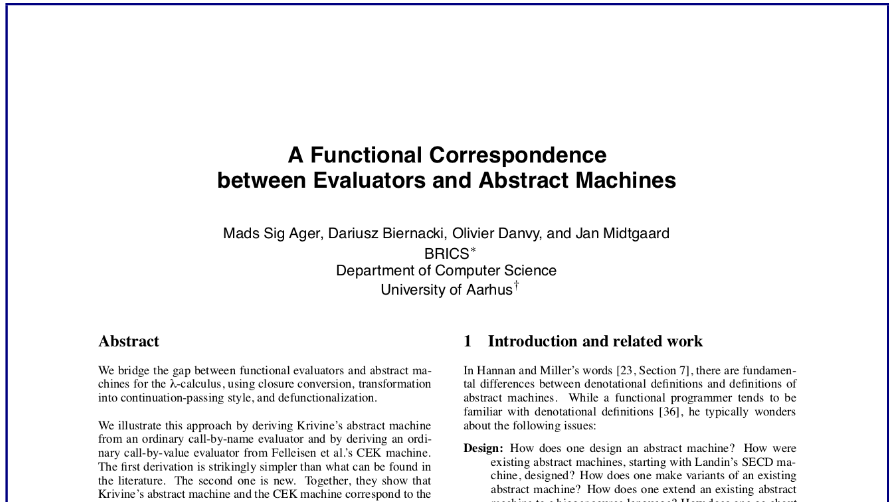

Definitional Interpreters for Higher-Order Programming Languages

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*
Department of Computer Science
University of Aarhus†

Abstract

We bridge the gap between functional evaluators and abstract machines for the λ-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