Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity

Jeremy Gibbons
University of Oxford

1. Factorial

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

Recursive.

Continuation-passing style

Introduce *continuation* as accumulating parameter:

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

Then calculate:

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

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

Defunctionalize

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

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

$$
\begin{aligned}
& fact_3 : Integer \to Integer \\
& fact_3\ n = fact'_3\ n\ [\,]\ \ \textbf{where} \\
& \quad fact'_3 : Integer \to List\ Integer \to Integer \\
& \quad fact'_3\ 0\ k = product\ k \\
& \quad fact'_3\ n\ k = fact'_3\ (n-1)\ (k \mathbin{+\!\!+} [\,n\,])
\end{aligned}
$$

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

Associativity

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

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

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

Data refinement valid by *associativity*.

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

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

2. Hutton's Razor

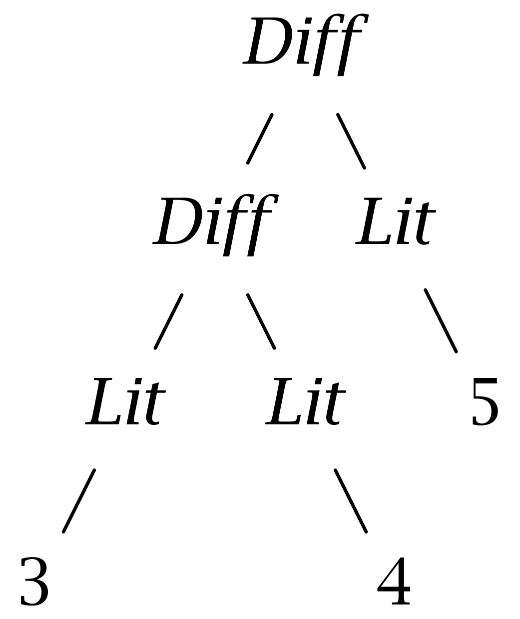
Expressions with subtraction, which is not associative:

$$\textbf{data}\ \mathit{Expr} = \mathit{Lit}\ \mathit{Integer} \mid \mathit{Diff}\ \mathit{Expr}\ \mathit{Expr}$$

$$\begin{aligned} &\mathit{expr} : \mathit{Expr} \\ &\mathit{expr} = \mathit{Diff}\ (\mathit{Diff}\ (\mathit{Lit}\ 3)\ (\mathit{Lit}\ 4))\ (\mathit{Lit}\ 5) \quad \text{-- ie } (3-4)-5 \end{aligned}$$

Evaluation:

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



CPS

$$
\begin{aligned}
& eval_2 : Expr \to Integer \\
& eval_2\ e = eval'_2\ e\ id\ \ \mathbf{where} \\
& \quad eval'_2 : Expr \to (Integer \to Integer) \to Integer \\
& \quad eval'_2\ (Lit\ n) \qquad\quad = \lambda k \Rightarrow k\ n \\
& \quad eval'_2\ (Diff\ e\ e') = \lambda k \Rightarrow eval'_2\ e\ (\lambda m \Rightarrow eval'_2\ e'\ (\lambda n \Rightarrow k\ (m - n)))
\end{aligned}
$$

Tail-recursive, but higher-order.

CPS with convenient abbreviations

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

Tail-recursive, but higher-order.

Abbreviations:

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

Defunctionalize

$$\textbf{data}\ \mathit{EvalFrame}_3 = \mathit{EvalLeftExpr}_3\ \mathit{Expr} \mid \mathit{EvalRightValue}_3\ \mathit{Integer}$$

$$
\begin{array}{l}
\mathit{eval}_3 : \mathit{Expr} \to \mathit{Integer} \\
\mathit{eval}_3\ e = \mathit{eval}'_3\ e\ [\,]\ \ \textbf{where mutual} \\
\quad \mathit{eval}'_3 : \mathit{Expr} \to \mathit{List}\ \mathit{EvalFrame}_3 \to \mathit{Integer} \\
\quad \mathit{eval}'_3\ (\mathit{Lit}\ n)\qquad k = \mathit{evalabs}_3\ k\ n \\
\quad \mathit{eval}'_3\ (\mathit{Diff}\ e\ e')\ k = \mathit{eval}'_3\ e\ (\mathit{EvalLeftExpr}_3\ e' :: k) \\
\quad \mathit{evalabs}_3 : \mathit{List}\ \mathit{EvalFrame}_3 \to (\mathit{Integer} \to \mathit{Integer}) \\
\quad \mathit{evalabs}_3\ [\,]\qquad\qquad\qquad\qquad n\ = n \\
\quad \mathit{evalabs}_3\ (\mathit{EvalLeftExpr}_3\ e' :: k)\qquad m = \mathit{eval}'_3\ e'\ (\mathit{EvalRightValue}_3\ m :: k) \\
\quad \mathit{evalabs}_3\ (\mathit{EvalRightValue}_3\ m :: k)\ n\ = \mathit{evalabs}_3\ k\ (m - n)
\end{array}
$$

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

Where does this compiler come from?

$\textbf{data}\ \mathit{Instr} = \mathit{PushI}\ \mathit{Integer} \mid \mathit{SubI}$

-- eg $[\mathit{PushI}\ 3, \mathit{PushI}\ 4, \mathit{SubI}, \mathit{PushI}\ 5, \mathit{SubI}]$ — ie *linear* code

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

$$\mathit{exec}_4 : \mathit{List}\ \mathit{Instr} \to \mathit{List}\ \mathit{Integer} \to \mathit{List}\ \mathit{Integer}$$
$$\mathit{exec}_4\ p\ s = \mathit{foldl}\ \mathit{step}\ s\ p\ \ \textbf{where}$$
$$\quad \mathit{step}\ \mathit{ns}\ (\mathit{PushI}\ n) = n :: \mathit{ns}$$
$$\quad \mathit{step}\ (n :: m :: \mathit{ns})\ \mathit{SubI} = (m - n) :: \mathit{ns}$$

-- note *reversal* of arguments

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

3. Generalized composition

Wand's key insight (1982). Recursive case routes k to $eval'_2\ e$, but k, m to $eval'_2\ e'$:

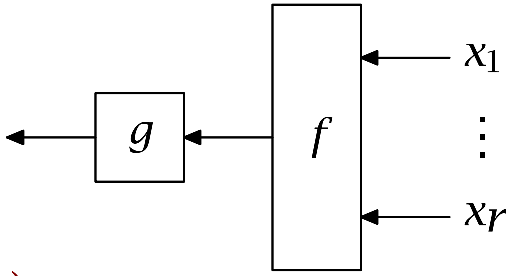
$$eval'_2\ (Diff\ e\ e') = \lambda k \Rightarrow eval'_2\ e\ (\lambda m \Rightarrow eval'_2\ e'\ (sub\ k\ m))$$

Generalize composition to propagate *multiple arguments*:

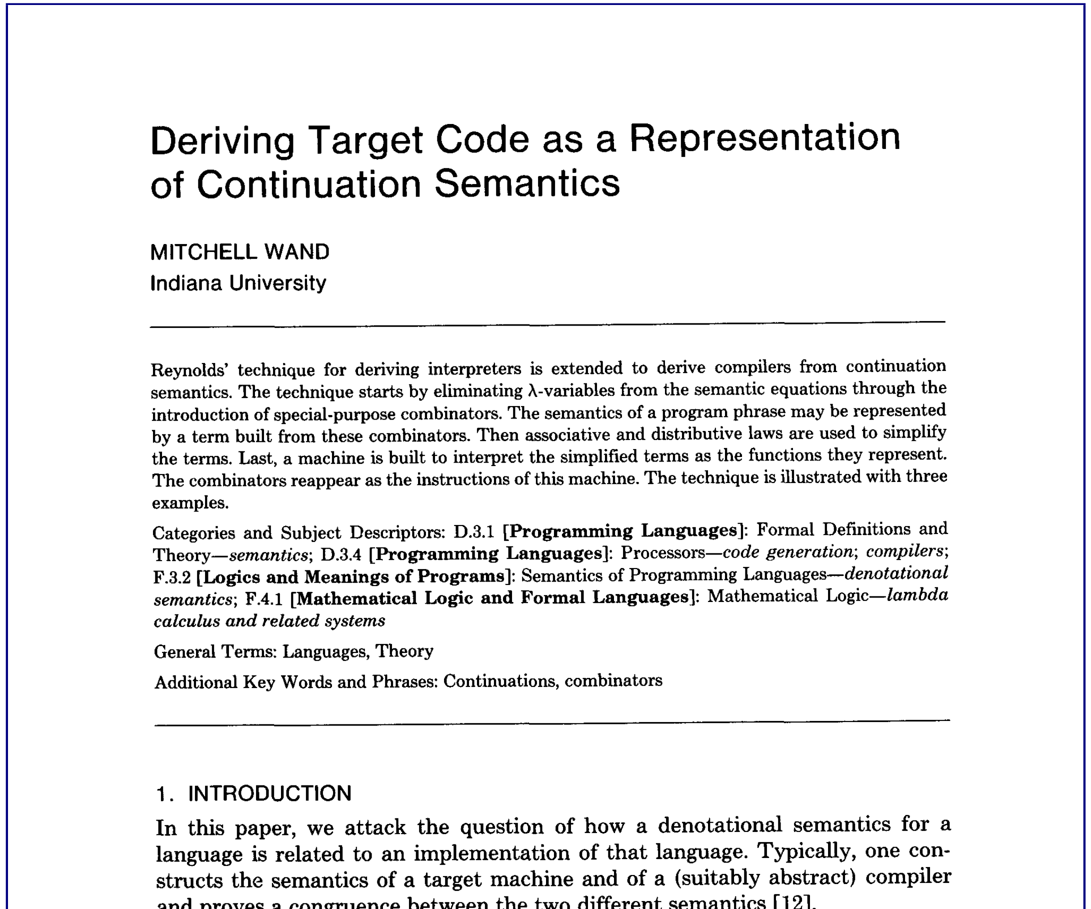
$$b^r\ g\ f = \lambda x_1 \ldots x_r \rightarrow g\ (f\ x_1 \ldots x_r)$$

ie

$$\begin{aligned} b^0\ g\ f &= g\ f \\ b^1\ g\ f &= g \cdot f \\ b^{r+1}\ g\ f &= \lambda x \rightarrow b^r\ g\ (f\ x) \end{aligned}$$



or equivalently, $b^r = (\cdot) \cdots (\cdot)$ (r times).



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

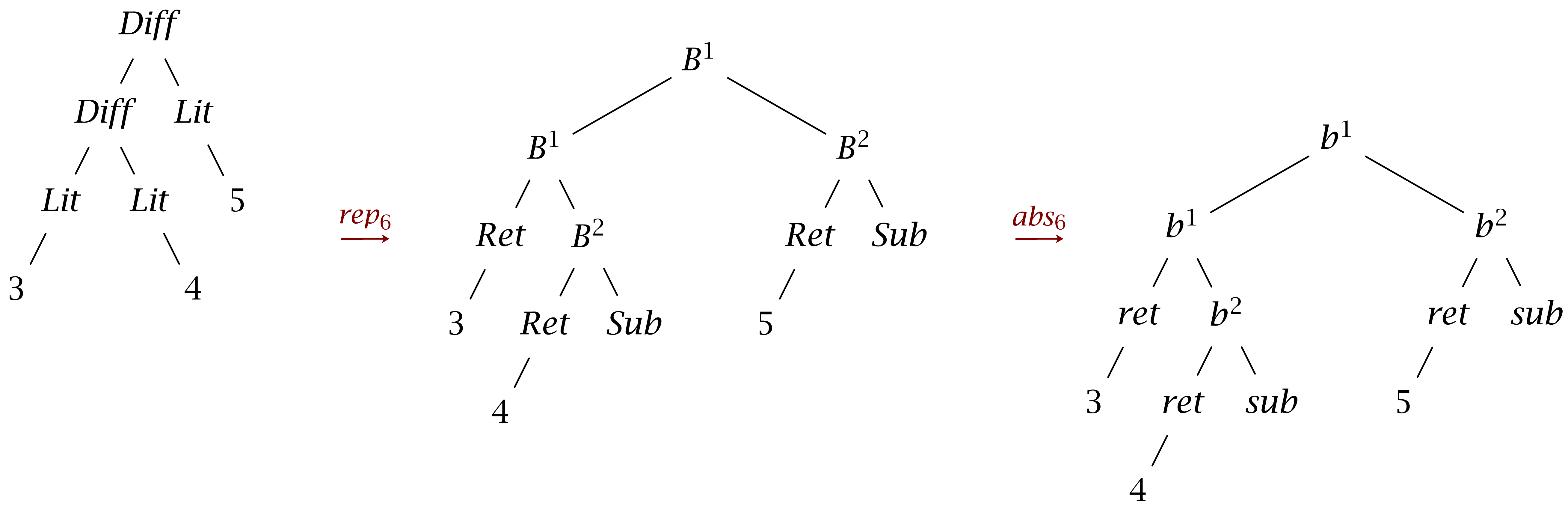
Additional Key Words and Phrases: Continuations, combinators

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].

Omitted steps (see paper)...

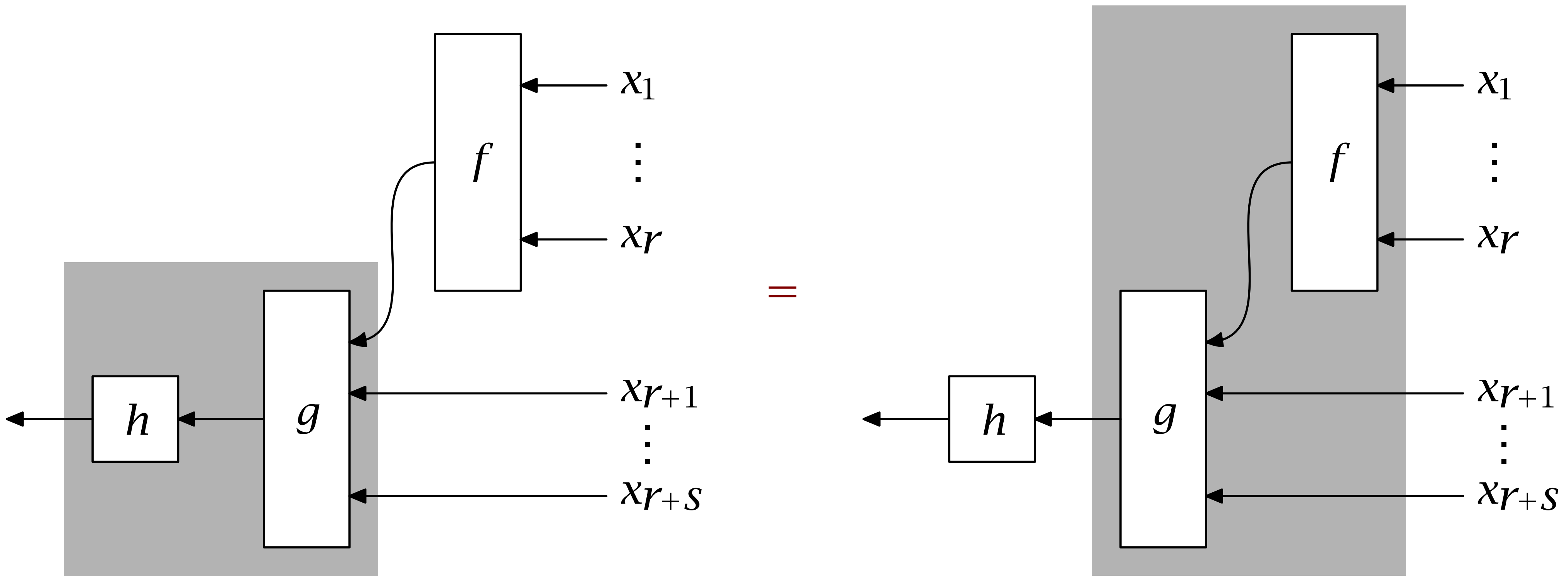
But still tree-shaped



How do we recover *linear* code?

4. Associativity

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



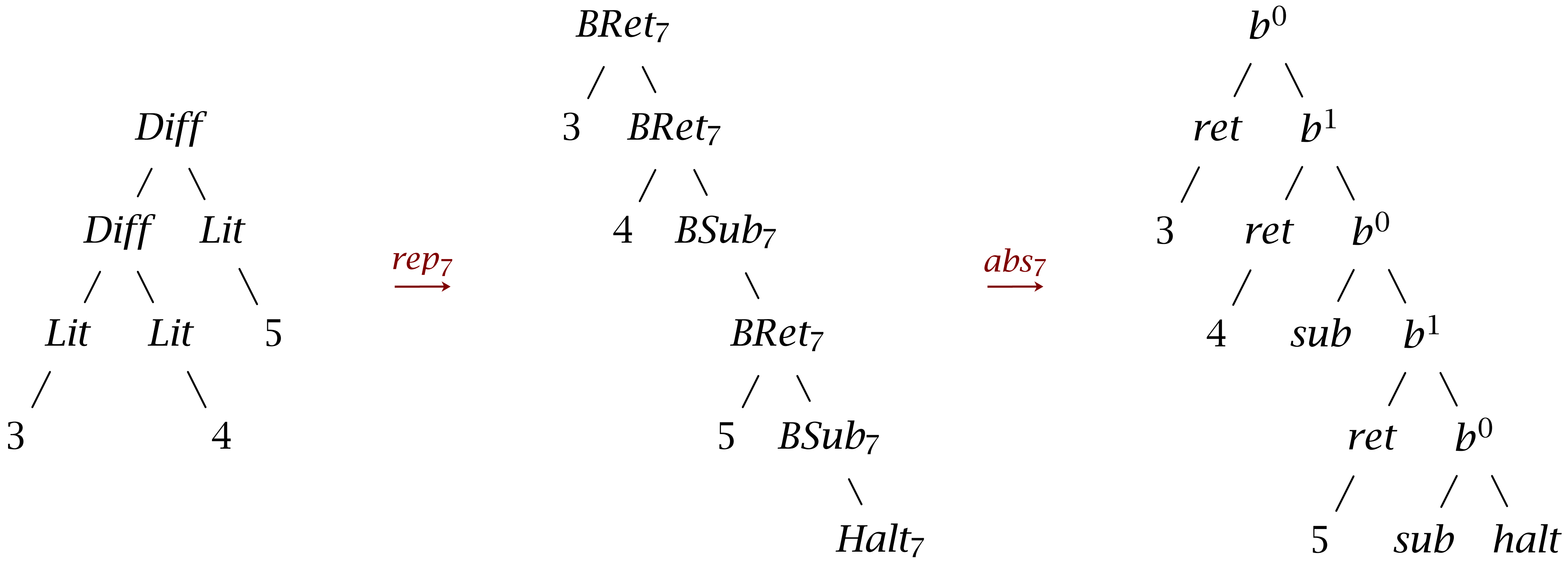
ie $b^r\ (b^{s+1}\ h\ g)\ f = b^{r+s}\ h\ (b^r\ g\ f)$. So we can *rotate* tree-shaped code to linear.

Rotating

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

More omitted steps (see paper)...

No longer tree-shaped



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

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

Indeed:

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

5. Conclusion

- *accumulating parameters*, *continuation-passing style*, *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
- *generalized composition* a useful tool
- perhaps it boils down to Cayley's Theorem / *Yoneda Lemma*?

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

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

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.

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

TFP (and TFPiE) 2025

26th International Symposium on Trends in Functional Programming
13th to 16th January 2025, Oxford, UK

The symposium on Trends in Functional Programming (TFP) is an international forum for researchers with interests in all aspects of functional programming, taking a broad view of current and future trends in the area. It aspires to be a lively environment for presenting the latest research results, and other contributions. See the call for papers for more details.

In 2025, the event is taking place in person in the Department of Computer Science at the University of Oxford. It will be a 4-day event, with TFPiE taking place on 13th January 2025, followed by TFP on 14th to 16th January.

TFP offers a friendly and constructive reviewing process designed to help less experienced authors succeed, with an opportunity for two rounds of review, both before and after the symposium itself. Authors thus have an opportunity to address reviewers' concerns before the final decision on publication in the Proceedings is taken, in the light of previous reviews and discussions at the symposium.

Omitted material

Implementing generalized composition

Really needs a *dependent type*, indexed by list of argument types:

$$\begin{aligned}
&\mathit{Arrow} : \mathit{List}\ \mathit{Type} \to \mathit{Type} \to \mathit{Type} \\
&\mathit{Arrow}\ [\,]\qquad\quad b = b \\
&\mathit{Arrow}\ (a :: as)\ b = a \to \mathit{Arrow}\ as\ b
\end{aligned}$$

Idris

For example, at arity 2:

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

Then defined by induction over the arity:

$$\begin{aligned}
&b : \{as : \mathit{List}\ \mathit{Type}\} \to (b \to c) \to \mathit{Arrow}\ as\ b \to \mathit{Arrow}\ as\ c \\
&b\ \{as = [\,]\}\qquad g\ f = g\ f \\
&b\ \{as = _ :: _\}\ g\ f = b\ g \cdot f
\end{aligned}$$

Exploiting generalized composition

Recall:

$$
\begin{aligned}
&eval_2\ e = eval_2'\ e\ id\\
&eval_2'\ (Lit\ n) \qquad = \lambda k \Rightarrow k\ n\\
&eval_2'\ (Diff\ e\ e') = \lambda k \Rightarrow eval_2'\ e\ (\lambda m \Rightarrow eval_2'\ e'\ (\lambda n \Rightarrow k\ (m - n)))
\end{aligned}
$$

Exploiting generalized composition

Recall:

$$
\begin{aligned}
&\mathit{eval}_2\ e = \mathit{eval}'_2\ e\ \mathit{halt} \\
&\mathit{eval}'_2\ (\mathit{Lit}\ n) \qquad = \mathit{ret}\ n \\
&\mathit{eval}'_2\ (\mathit{Diff}\ e\ e') = \lambda k \Rightarrow \mathit{eval}'_2\ e\ (\lambda m \Rightarrow \mathit{eval}'_2\ e'\ (\lambda n \Rightarrow \mathit{sub}\ k\ m\ n))
\end{aligned}
$$

where for later convenience we introduce:

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

Exploiting generalized composition

Recall:

$$eval_2\ e = eval_2'\ e\ halt$$
$$eval_2'\ (Lit\ n) \qquad = ret\ n$$
$$eval_2'\ (Diff\ e\ e') = \lambda k \Rightarrow eval_2'\ e\ (\lambda m \Rightarrow eval_2'\ e'\ (\lambda n \Rightarrow sub\ k\ m\ n))$$

Then:

$$eval_2'\ (Diff\ e\ e')$$
$= \quad$ [[definition]]
$$\lambda k \Rightarrow eval_2'\ e\ (\lambda m \Rightarrow eval_2'\ e'\ (\lambda n \Rightarrow sub\ k\ m\ n))$$
$= \quad$ [[since $\lambda k \Rightarrow g\ (f\ k)$ is $b^1\ g\ (\lambda k \Rightarrow f\ k)$]]
$$b^1\ (eval_2'\ e)\ (\lambda k\ m \Rightarrow eval_2'\ e'\ (\lambda n \Rightarrow sub\ k\ m\ n))$$
$= \quad$ [[since $\lambda k\ m \Rightarrow g\ (f\ k\ m)$ is $b^2\ g\ (\lambda k\ m \Rightarrow f\ k\ m)$]]
$$b^1\ (eval_2'\ e)\ (b^2\ (eval_2'\ e')\ sub)$$

Installing generalized composition

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

$$
\begin{aligned}
&eval_5 : Expr \to Integer \\
&eval_5\ e = eval'_5\ e\ halt\ \ \mathbf{where} \\
&\quad eval'_5 : Expr \to (Integer \to Integer) \to Integer \\
&\quad eval'_5\ (Lit\ n) \qquad\ = ret\ n \\
&\quad eval'_5\ (Diff\ e\ e') = b^1\ (eval'_5\ e)\ (b^2\ (eval'_5\ e')\ sub)
\end{aligned}
$$

Representation

$eval'_5$ is *not tail-recursive* any more; but suggests another representation:

$$
\begin{aligned}
&\textbf{data}\ ExprRep_6 : List\ Type \to Type\ \ \textbf{where} \\
&\quad Ret_6 : Integer \to \qquad\qquad\qquad\qquad\qquad\qquad\qquad ExprRep_6\ [\,] \\
&\quad Sub_6 : \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ExprRep_6\ [Integer, Integer] \\
&\quad B^1_6 : ExprRep_6\ [\,] \to ExprRep_6\ [Integer] \to \qquad\qquad\quad ExprRep_6\ [\,] \\
&\quad B^2_6 : ExprRep_6\ [\,] \to ExprRep_6\ [Integer, Integer] \to ExprRep_6\ [Integer]
\end{aligned}
$$

obtained by *defunctionalizing the evaluator*:

$$
\begin{aligned}
&rep_6 : Expr \to ExprRep_6\ [\,] \\
&rep_6\ (Lit\ n) \qquad = Ret_6\ n \\
&rep_6\ (Diff\ e\ e') = B^1_6\ (rep_6\ e)\ (B^2_6\ (rep_6\ e')\ Sub_6)
\end{aligned}
$$

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

Interpretation

Data of type $\mathit{ExprRep}_6\ r$ is a defunctionalized evaluation function of type

$$(\mathit{Integer} \to \mathit{Integer}) \to \mathit{Arrow}\ r\ \mathit{Integer}$$

Abstraction function *refunctionalizes*:

$$
\begin{aligned}
&\mathit{abs}_6 : \mathit{ExprRep}_6\ r \to (\mathit{Integer} \to \mathit{Integer}) \to \mathit{Arrow}\ r\ \mathit{Integer} \\
&\mathit{abs}_6\ (\mathit{Ret}_6\ n) = \mathit{ret}\ n \\
&\mathit{abs}_6\ \mathit{Sub}_6 \qquad = \mathit{sub} \\
&\mathit{abs}_6\ (B^1_6\ x\ y) = b^1\ (\mathit{abs}_6\ x)\ (\mathit{abs}_6\ y) \\
&\mathit{abs}_6\ (B^2_6\ x\ y) = b^2\ (\mathit{abs}_6\ x)\ (\mathit{abs}_6\ y)
\end{aligned}
$$

Linear code

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

supporting concatenation:

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

Representation and interpretation

Obtained by defunctionalizing the transformed interpreter:

$$
\begin{aligned}
&rep_7 : Expr \rightarrow ExprRep_7\ [\,]\\
&rep_7\ (Lit\ n) \quad\ = BRet_7\ n\ Halt_7\\
&rep_7\ (Diff\ e\ e') = append_7\ (rep_7\ e)\ (append_7\ (rep_7\ e')\ (BSub_7\ Halt_7))
\end{aligned}
$$

and interpreted like this:

$$
\begin{aligned}
&abs_7 : ExprRep_7\ r \rightarrow Arrow\ r\ Integer\\
&abs_7\ Halt_7 \qquad\quad = halt\\
&abs_7\ (BRet_7\ n\ k) = ret\ n\ (abs_7\ k)\\
&abs_7\ (BSub_7\ k) \quad\ = flip\ (sub\ (abs_7\ k)) \qquad \text{-- note } \textit{reversal} \text{ of arguments again}
\end{aligned}
$$