No-Brainer CPS Conversion (Functional Pearl)

MILO DAVIS, WILLIAM MEEHAN, and OLIN SHIVERS, Northeastern University, USA

Algorithms that convert direct-style λ-calculus terms to their equivalent terms in continuation-passing style (CPS) typically introduce so-called "administrative redexes:" useless artifacts of the conversion that must be cleaned up by a subsequent pass over the result to reduce them away. We present a simple, linear-time algorithm for CPS conversion that introduces no administrative redexes. In fact, the output term is a normal form in a reduction system that generalizes the notion of "administrative redexes" to what we call "no-brainer redexes," that is, redexes whose reduction shrinks the size of the term. We state the theorems which establish the algorithm's desireable properties, along with sketches of the full proofs.

CCS Concepts: • **Software and its engineering → Functional languages**; **Syntax**; **Source code generation**; **Preprocessors**;

Additional Key Words and Phrases: continuation-passing style, CPS, functional language, continuation, lambda calculus, compiler

ACM Reference Format:
Milo Davis, William Meehan, and Olin Shivers. 2017. No-Brainer CPS Conversion (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 23 (September 2017), 25 pages.
https://doi.org/10.1145/3110267

1 INTRODUCTION

Continuation-Passing Style (CPS) is a restricted form of the λ-calculus that has many pleasant properties for both theoretical and practical use [Appel 2006; Fischer 1993; Kelsey 1989, 1995; Kelsey and Hudak 1989; Kennedy 2007; Kranz et al. 1986; Kranz 1988; Plotkin 1975; Reynolds 1972, 1993; Sabry and Felleisen 1993; Steele Jr. 1976, 1978]. However, despite all the positive traits one might associate with the form, not many would assert it is pleasant to write: CPS is a low-level form more suited as a kind of intermediate representation than as something directly written by a human. Typically, we write terms in plain, old "direct-style" λ-calculus, and then run the term through a "CPS converter" algorithm or function that does the translation for us.

One of the undesirable traits of CPS-conversion algorithms is that they can introduce unwanted "administrative reductions" that have to be cleaned up afterward by a post-pass over the result. These redexes are called "administrative" because they are introduced during the transformation and are not present in the original program. These redexes are ones that are conservatively introduced by the algorithm in its efforts to provide every intermediate quantity computed by the original term with a name—even terms that are *already* named (that is, variable references), or terms that don't really need to be named (such as λ-terms that are only going to be used in one place).

2475-1421/2017/9-ART23
https://doi.org/10.1145/3110267

This issue can be seen when converting a simple direct-style (DS) term using the classic Fischer/Reynolds algorithm [Fischer 1993; Reynolds 1972]. If we start with the direct-style

$$(\lambda x.x)\ (\lambda x.x),$$

the Fischer/Reynolds algorithm gives us

$$\begin{aligned}&\lambda k_1.(\lambda k_2.k_2(\lambda x.\lambda k_3.k_3 x))\\&\qquad(\lambda m.(\lambda k_4.k_4(\lambda x.\lambda k_5.k_5 x))\\&\qquad\quad \lambda n.m\ n\ k_1).\end{aligned}$$

Compare this messy output with the simpler result produced by the later Danvy/Filinski algorithm [1992] or the equivalent Sabry/Felleisen algorithm [1993]:

$$\begin{aligned}&\lambda k_1.(\lambda x.\lambda k_3.k_3 x)\\&\qquad(\lambda x.\lambda k_5.k_5 x)\\&\qquad k_1.\end{aligned}$$

The history of CPS conversion has seen a mini arms-race in the production of conversion algorithms that produce terms with fewer and fewer of these annoying and useless redexes. But there is still room for improvement.

With this pearl, we'd like to get away from the very concept of "administrative redexes," and simply consider, more generally, the idea of "redexes we obviously don't want." That is, redexes whose reduction is a "no brainer."

In this pearl, we'll define precisely what we mean by "no brainer" redexes, and then develop a simple CPS-conversion algorithm that produces a result that does not contain, at all, three of the four kinds of these redexes. We stay within the classical rules of the game for previous CPS-conversion algorithms: the converter runs in linear time, and works by means of simple, recursive tree-walks of the original term. Our algorithm produces a result that is smaller and simpler than any alternative, linear-time algorithm of which we know. A complete implementation, shown in an appendix, is 101 lines of OCaml code (not counting blank lines and comments), of which 22 lines are type definitions, and the remaining 79 lines, actual code.

To return to our running example, if we convert the direct-style term above using our algorithm, we get

$$\lambda k_1.k_1(\lambda x.\lambda k_5.k_5 x).$$

2 NOTATION

We use "Scott brackets" $[\![\cdot]\!]$ in order to write down abstract-syntax elements using concrete-syntax form [Rabern 2016]. For example, when we write "`(if x 3 (+ 2 5))`," we mean the literal string whose first few characters are left-parenthesis, i, f, and so on. However, when we enclose this string in doubled brackets, $[\![$`(if x 3 (+ 2 5))`$]\!]$, we mean a mathematical element of a domain of abstract syntax. This domain is an inductively defined set of syntax trees; we are now in the realm of structure (trees), not concrete syntax (strings of characters). In our example, the particular element we are describing is a conditional expression that has three child expressions: a reference to the variable `x`, the numeric constant three, and an application of the addition function to the constants two and five. Again, double brackets lets us write down the abstract value using the concrete, surface notation for the element.

We extend this notation by permitting ourselves to insert subterms of abstract syntax, written in italicised, math notation, in the midst of one of these strings of concrete syntax. For example, if we let $v = [\![\texttt{x3}]\!]$, then $[\![\texttt{(if (< } v \texttt{ 0) (- } v\texttt{) } v\texttt{)}]\!]$ is the Lisp expression computing the absolute value of the number stored in the Lisp variable `x3`, all represented as a tree of abstract syntax. In particular,

$y \in$ DSVAR			*DS Variable*
$fun \in$ FUN	::=	`(fun` y e`)`	*λ-term*
$e \in$ DS	::=	y	*Var reference*
	|	fun	*Abstraction*
	|	`(`e e`)`	*Application*
	|	`(if` e e e`)`	*Conditional*

(a) Direct-style syntax

$x \in$ UVAR			*User variable*
$k \in$ CVAR			*Cont variable*
$lam \in$ LAM	::=	`(lam (`x k`)` p`)`	*User λ-term*
$clam \in$ CLAM	::=	`(cont` x p`)`	*Cont λ-term*
$triv \in$ TRIV	::=	x | lam	*User argument*
$cont \in$ CONT	::=	k | $clam$ | `halt`	*Cont argument*
$p \in$ CPS	::=		*CPS program*
		`(call` $triv$ $triv$ $cont$`)`	*Function app*
	|	`(ret` $cont$ $triv$`)`	*Cont app*
	|	`(if` $triv$ p p`)`	*Conditional*
	|	`(letc (`k $cont$`)` p`)`	*Cont binding*

(b) CPS syntax

Fig. 1. Syntax of the source and target languages

note that the variable v is a *metasyntactic*, *math* variable, whose value is the Lisp variable `x3`. We will sometimes omit these brackets in top-level cases when there is danger of no confusion.

(Readers comfortable with the Lisp "backquote" notation will not go far wrong by interpreting double-brackets as backquote and italicised mathematical material as being comma-prefixed.)

3 LANGUAGES

We use an s-expression syntax for our terms; and we've made the direct-style source language (Figure 1(a)) syntactically distinct from the CPS target language (Figure 1(b)). Using s-expressions makes a clear distinction between meta-syntactic, mathematical expressions and the actual terms we are processing.

We've included a primitive conditional `if` form to exercise the conversion algorithm in cases where a continuation is shared by both arms of the conditional—CPS algorithms have to be careful not to duplicate code when they do this sharing, else they could induce exponential blowup in the size of the result term.

We've also taken the trouble to use a "factored" CPS language, where continuation functions, applications, and variable references are syntactically distinct from "user" terms that are non-continuations. So, for example, a "user" function is written `(lam (`x k`)` p`)`; it binds user parameter x and continuation parameter k, and is applied with a `call` form. A continuation function is written `(cont` x p`)`, and is applied with a `ret` form.

User arguments *triv* are traditionally referred to as "trivial" arguments because they are limited to variable references and λ terms. This means that, unlike argument expressions e in the direct-style language, they can always be trivially evaluated to a value in constant time, with no side effects, control or otherwise. A pleasant consequence of this property is that β-reduction is *always* sound in CPS.

The `letc` form is for let-binding a continuation form to a continuation variable. In our simple, core language, we can't use a β-redex to do this, since the only other form that binds a continuation variable is the `lam` form, which *also* binds a user variable. (We discuss more practical engineering alternatives in Section 7. Going with `letc` lets us keep things simple for expository purposes.)

We use the function $nref(x, e)$ to mean the number of references to variable x that occur free in expression e, and the function $FV(e)$ to mean the set of variables occurring free in expression e.

4 THE NO-BRAINER REDUCTION SYSTEM

When we set about simplifying a large, complex term written out in the λ-calculus, it's just obvious that some redexes should be simplified away—it's a "no-brainer" decision to reduce them. What constitutes such a redex? It is *a redex whose contraction will immediately reduce the size of the program.*

Note the condition that the size reduction be immediate: there may be other redexes whose contraction may locally *expand* the program, but this expansion will introduce new reduction opportunities that will eventually lead to an overall, global decrease in the size of the program. We aren't considering these reductions, which require search and complex analysis to find. Rather, we conservatively restrict ourselves to greedy, hill-climbing optimisations. The charm of such improvements is that they are easily detected from local context, and they are always a good idea—that's what makes the decision to perform them a no-brainer.

Here are four kinds of redex that are all guaranteed to shrink the size of the program in which they occur:

(1) A β-redex where the bound variable has multiple references, but the substituend (the value being substituted for the bound variable) is of unit size and hence can be swapped in for the multiple references without increasing term size. A "unit-size" substituend is either a variable or a small constant, such as an integer (as opposed to a large constant that we don't want to replicate, such as a large constant list structure or other aggregate data structure). For example:

```
(ret (cont x
        ... x ... x ...)   →   ... y ... y ...
     y)
```

(2) A β-redex where the bound variable being substituted away has exactly one reference in the body of its binding λ-term. For example:

```
(ret (cont x
        ... x ...)   →   ... triv ...
     triv)
```

(3) A β-redex where the bound variable being substituted away has no references at all in the body of its binding λ-term, *e.g.*,

```
(ret (cont x body)
     triv)            →   body
```

(4) An η-redex, *e.g.*, `(lam (x k) (call triv x k))` $\rightarrow$ *triv*.

Note that these reductions might not be semantics-preserving in a direct-style term with call-by-value semantics, but they are always legal in a CPS term, due to the fact that substituends are effect-free trivial arguments.

Our algorithm handles three of these four reductions; the one we cannot handle is #3, where the bound variable has zero references. Reduction #3 is problematic because it causes the reference counts of other variables to be lowered. Doing such a reduction essentially deletes the substituend, which might be a large term containing many free-variable references. Deleting the term means deleting these references, which has an "action at a distance" effect: by lowering the reference counts of other variables, we might make other redexes suddenly become redexes of types #2 or #3. We could not find a way to manage this sort of cascading of reduction opportunity in a simple tree walk.

If we take the remaining three kinds of no-brainer redexes and apply them to any subterm of some large CPS program, we get a rewrite system we call the "No-brainer Reduction" (NBR) system. The NBR system is the specification of what we are trying to accomplish when we set out to clean up the output of a CPS conversion.

The point of the NBR system is that we don't care about the *source* of No-Brainer redexes; we just want them gone—all of them. Attributing some of these redexes to the conversion algorithm, which is what we do when we label them "administrative," is just a distraction. Furthermore, it's a questionable attribution. All redexes in the output of a CPS conversion were introduced by the converter—the ones we want to keep, and the ones we want to eliminate! So, in this sense, our goal is more ambitious than eliminating so-called "administrative" redexes: our goal is a clean, simplified result term.

4.1 Formalising NBR

We define NBR to consist of the following reductions:

Definition 4.1 (β_{cv}). β-reduction where the substituend is a variable, or a unit-sized constant:

$$\begin{gathered}
\texttt{(call (lam (}x_1\ k\texttt{) p) }x_2\ \mathit{cont}\texttt{)} \xrightarrow[\beta cv]{} \texttt{(letc (}k\ \mathit{cont}\texttt{) }p[x_1 \mapsto x_2]\texttt{)}\\
\texttt{(call (lam (}x\ k_1\texttt{) p) }\mathit{triv}\ k_2\texttt{)} \xrightarrow[\beta cv]{} \texttt{(ret (cont }x\ p[k_1 \mapsto k_2]\texttt{) }\mathit{triv}\texttt{)}\\
\texttt{(call (lam (}x\ k\texttt{) }p\texttt{) }\mathit{triv}\ \texttt{halt)} \xrightarrow[\beta cv]{} \texttt{(ret (cont }x\ p[k \mapsto \texttt{halt}]\texttt{) }\mathit{triv}\texttt{)}\\
\texttt{(ret (cont }x_1\ p\texttt{) }x_2\texttt{)} \xrightarrow[\beta cv]{} p[x_1 \mapsto x_2]\\
\texttt{(letc (}k_1\ k_2\texttt{) }p\texttt{)} \xrightarrow[\beta cv]{} p[k_1 \mapsto k_2]\\
\texttt{(letc (}k\ \texttt{halt) }p\texttt{)} \xrightarrow[\beta cv]{} p[k \mapsto \texttt{halt}]
\end{gathered}$$

Clearly, replacing one variable with another will not change the term's size. We also can inline constants with unit size, such as the top-level `halt` continuation.

Definition 4.2 ($\beta_{\lambda 1}$). β-reduction where the substituend is a λ-term (*i.e.*, a `lam` or `cont` form) and the bound variable is referenced exactly once in the redex:

$$\begin{gathered}
\texttt{(call (lam (}x_1\ k\texttt{) }p\texttt{) }\mathit{lam}\ \mathit{cont}\texttt{)} \xrightarrow[\beta\lambda 1]{} \texttt{(letc (}k\ \mathit{cont}\texttt{) }p[x_1 \mapsto \mathit{lam}]\texttt{)}\\
\texttt{(call (lam (}x\ k_1\texttt{) }p\texttt{) }\mathit{triv}\ \mathit{clam}\texttt{)} \xrightarrow[\beta\lambda 1]{} \texttt{(ret (cont }x\ p[k_1 \mapsto \mathit{clam}]\texttt{) }\mathit{triv}\texttt{)}\\
\texttt{(ret (cont }x_1\ p\texttt{) }\mathit{lam}\texttt{)} \xrightarrow[\beta\lambda 1]{} p[x_1 \mapsto \mathit{lam}]\\
\texttt{(letc (}k_1\ \mathit{clam}\texttt{) }p\texttt{)} \xrightarrow[\beta\lambda 1]{} p[k_1 \mapsto \mathit{clam}]
\end{gathered}$$

where $\mathit{nref}(k_1,p) = 1$ and $\mathit{nref}(x_1,p) = 1$.

$$
\begin{array}{lcl}
C_p & ::= & \Box^p \\
 & | & \texttt{(call } C_t \ \mathit{triv}\ \mathit{cont}\texttt{)} \mid \texttt{(call } \mathit{triv}\ C_t\ \mathit{cont}\texttt{)} \\
 & | & \texttt{(call } \mathit{triv}\ \mathit{triv}\ C_c\texttt{)} \\
 & | & \texttt{(ret } C_c\ \mathit{triv}\texttt{)} \mid \texttt{(ret } \mathit{cont}\ C_t\texttt{)} \\
 & | & \texttt{(if } C_t\ p_1\ p_2\texttt{)} \\
 & | & \texttt{(if } \mathit{triv}\ C_p\ p\texttt{)} \mid \texttt{(if } \mathit{triv}\ p\ C_p\texttt{)} \\
 & | & \texttt{(letc (}k\ C_c\texttt{)}\ p\texttt{)} \mid \texttt{(letc (}k\ \mathit{cont}\texttt{)}\ C_p\texttt{)} \\
C_t & ::= & \Box^t \mid \texttt{(lam (}x\ k\texttt{)}\ C_p\texttt{)} \\
C_c & ::= & \Box^c \mid \texttt{(cont } x\ C_p\texttt{)}
\end{array}
$$

(a) CPS reduction-context grammar

$$
\frac{p_1 \to p_2}{C_p^p\,[p_1] \to C_p^p\,[p_2]}
\qquad
\frac{\mathit{triv}_1 \to \mathit{triv}_2}{C_p^t\,[\mathit{triv}_1] \to C_p^t\,[\mathit{triv}_2]}
\qquad
\frac{\mathit{cont}_1 \to \mathit{cont}_2}{C_p^c\,[\mathit{cont}_1] \to C_p^c\,[\mathit{cont}_2]}
$$

(b) Reduction inference rules

Fig. 2. This context grammar captures the idea that we want to permit No-Brainer reductions anywhere they occur in a program.

We also want to inline `lam` and `cont` function abstractions. In the best case, this can create more potential redexes, leading to cascades of inlining. However, to prevent code growth, we restrict the $\beta_{\lambda 1}$ redex, requiring that the variable being substituted has only one reference in the body of the binding λ-term.[1]

Definition 4.3 (η). Classic η-reduction:

$$
\begin{array}{rcl}
\texttt{(lam (}x\ k\texttt{) (call } \mathit{triv}\ x\ k\texttt{))} \underset{\eta}{\longrightarrow} \mathit{triv} & & (x, k \notin FV(\mathit{triv})) \\
\texttt{(cont } x\ \texttt{(ret } \mathit{cont}\ x\texttt{))} \underset{\eta}{\longrightarrow} \mathit{cont} & & (x \notin FV(\mathit{cont}))
\end{array}
$$

Finally, η-reduction always decreases term size. Note that, unlike β_{cv} and $\beta_{\lambda 1}$, η is not a reduction on programs, but rather on function abstractions.

Figure 2 shows a context grammar spelling out that we want to permit No-Brainer reductions anywhere they occur. A context can have one of three different kinds of hole, reflecting that we can plug in three different kinds of term: "program" terms ($\Box^p$), trivial user values ($\Box^t$), and continuation values ($\Box^c$). We use a subscript on a context to indicate what type of term results from plugging the hole. All top-level contexts must be program contexts, but because β-redexes are programs and η-redexes are either trivial arguments or continuations, we allow holes for any type of term. We can plug any p into a C_p^p, any *triv* into a C_p^t, and any *cont* into a C_p^c; all three of these will produce a "program" term.

If we permit β_{cv}, $\beta_{\lambda 1}$, and η reductions to occur anywhere the redexes occur as subterms of a CPS program, as expressed by the context grammar of Figure 2, we get the NBR system.

4.2 Properties of the NBR System

The NBR system has two nice properties. First, it is strongly normalising. It is easy to see that every No-Brainer reduction sequence on a CPS term must eventually terminate in a normal form, since every reduction step we take decreases the size of the term.

It's less obvious, but also true, that the system is confluent. Thus, if a term has multiple possible rewrites, we can do them in any order we like: we will always arrive at the same normal form. So

[1] We can easily extend this rule also to permit reduction when the substituend is a "large," non-unit-sized constant, such as constant list structure, that should not be replicated. We haven't shown this, as our simple source language doesn't have such constants. The key idea of this rule, in any event, is to express substitutions of large things for single-referenced variables.

$$F\,e\,cont \triangleq \begin{cases} \llbracket\texttt{(ret}\ cont\ e\texttt{)}\rrbracket & e \in \text{DSVAR} \\ \llbracket\texttt{(ret}\ cont\ \texttt{(lam (}y\ k\texttt{)}\ \texttt{(}F\,e'\,k\texttt{)))}\rrbracket & e = \llbracket\texttt{(fun}\ y\ e'\texttt{)}\rrbracket \\ F\,e_f \left\llbracket \begin{array}{l}\texttt{(cont}\ x_f \\ \quad \texttt{(}F\,e_a\,\llbracket\texttt{(cont}\ x_a\ \texttt{(call}\ x_f\ x_a\ cont\texttt{))}\rrbracket\texttt{))}\end{array}\right\rrbracket & e = \llbracket\texttt{(}e_f\ e_a\texttt{)}\rrbracket \\ F\,e_1 \left\llbracket \begin{array}{l}\texttt{(cont}\ x_b\ \texttt{(letc (}j\ cont\texttt{)} \\ \qquad\qquad \texttt{(if}\ x_b\ \texttt{(}F\,e_2\,j\texttt{)}\ \texttt{(}F\,e_3\,j\texttt{))))}\end{array}\right\rrbracket & e = \llbracket\texttt{(if}\ e_1\ e_2\ e_3\texttt{)}\rrbracket \end{cases}$$

Fig. 3. The Fischer/Reynolds algorithm

we can speak of *the* NB normal form of a term, rather than *a* NB normal form; we call this the "No-Brainer normal form" (NBNF) of the term.

In Subsection 6.1, we present sketches of proofs for both of these properties; the full proofs exist and are being included in a technical report under preparation.

4.3 The NBR System as a Specification

The real purpose of laying out the details of the No-Brainer reduction system is that it serves as a formal specification for the algorithm we'd like to develop. If we can show that some linear-time CPS-conversion algorithm produces a No-Brainer normal form, then we have reason to believe that we've gotten all the easy simplifications. The only ones we might have missed are the zero-reference reductions that require us to hop around the term as reference counts decrement due to other reductions, something that rules out a simple, linear-time, tree-walk traversal of the term.

5 THE ALGORITHM

Figure 3 gives our starting point, the Fischer/Reynolds algorithm [Fischer 1993; Reynolds 1972]. We'll adopt the convention here, and in later algorithms, that variables otherwise unspecified in result terms are taken to be fresh. The goal for our final, "smart" algorithm is to produce the term we would get if we first CPS converted a source term with the Fischer/Reynolds algorithm, then reduced it to a No-Brainer normal form—but we want to *directly* produce the final normal form, without ever creating any redexes which must be contracted away:

$$\begin{tikzcd} \text{DS} \arrow[d, "F/R"'] \arrow[dr, "Smart"] & \\ \text{CPS} \arrow[r, "NBR"] & \text{CPS/NBNF} \end{tikzcd}$$

5.1 Symbol Tables and Abstract-Syntax Domains

We will get from the Fischer/Reynolds algorithm to our final algorithm in two steps, first presenting a "dumb" algorithm as an intermediate waypoint. The Dumb algorithm produces the same output (up to α equivalence) as the Fischer/Reynolds algorithm, but has the structural elements we'll need for the smart algorithm. It has three principal distinctions from the Fischer/Reynolds algorithm. The first is that we introduce a symbol table, or static environment, which we need if we are taking our source/target language distinction seriously. After all, source-language variables come from a different syntactic domain (DSVAR) than target-language variables (UVAR). Whenever we translate

$$
\begin{array}{rcll}
c \in \text{ABS-CONT} & ::= & \texttt{halt} & \\
& | & k & \\
& | & \mathit{FCont}(e, E, c) & \texttt{(cont}\ x_f\ \texttt{...)} \\
& | & \mathit{ACont}(a, c) & \texttt{(cont}\ x_a\ \texttt{...)} \\
& | & \mathit{ICont}(e_1, e_2, E, c) & \texttt{(cont}\ x_b\ \texttt{(letc (}j\ \mathit{cont}\texttt{) (if}\ x_b\ \texttt{...)))} \\
& | & & \\
a \in \text{ABS-ARG} & ::= & x & \\
& | & \langle [\![\texttt{(fun}\ y\ e\texttt{)}]\!], E \rangle &
\end{array}
$$

Fig. 4. The *Abstract Continuation* data structure lets us delay the decision about whether we should directly produce a term for a continuation, or reduce it away. The three constructors correspond to continuations introduced by the conversion algorithm itself; the right column gives the syntactic term from the Fischer/Reynolds algorithm to which these items correspond (see Figure 3). Each constructor packages up enough information to produce the continuation it represents. Similarly, the *Abstract Argument* data structure represents CPS *triv* arguments, and lets us delay the convert-or-reduce decision for `fun` terms. We refer to $\langle \mathit{fun}, E\rangle$ code/environment pairs as "static closures."

a source `(fun` y e`)` term, the converter picks a fresh target-language variable x for the corresponding parameter in the result CPS `(lam (`x k`) ...)` term, and then CPS-converts the body e with a static environment that includes a $y \mapsto x$ mapping. Whenever we translate a source-language variable reference y, we look it up in the current static environment to find its corresponding target-language entry.

The second change is that we introduce *abstract continuations* and *abstract arguments*, defined in Figure 4. Let's consider abstract continuations first. There are five kinds: the primitive `halt` continuation, continuation terms that are variables, and the ones built with the three constructors, *FCont*, *ACont*, and *ICont*, which represent explicit continuation terms introduced by the converter algorithm. Here is an informal, English description of each of these three kinds of continuation:

- $\mathit{FCont}(e_a, E, c)$
  This continuation arises when we translate a direct-style function-application term `(`e_f e_a`)`. It is the continuation that is awaiting the value produced by the evaluation of (the CPS translation of) the application's *function* part, e_f. Thus, its mission in life could be stated as follows: "We have just received some function value. Now evaluate the application's argument part e_a, then apply the function value to the argument value, with the result of the function call going to continuation c." This is the continuation `(cont` x_f `...)` we see in the third case of the Fischer/Reynolds algorithm.
  It's important to note that this abstract continuation is constructed from the *source* term e_a, so we must package up a static environment E along with this term—when we eventually get around to rendering this abstract continuation into concrete syntax, we'll have to CPS convert e_a, and so will need E to tell us how to translate the free-variable references occuring in e_a.
- $\mathit{ACont}(a_f, c)$
  This is the continuation that is awaiting the result of evaluating the *argument* part of a direct-style function application; a_f is an abstract trivial-argument term representing the value that was produced when we earlier evaluated the application's function term. Thus, this continuation's mission in life is: "apply a_f to the value we have just received, with the result of the application going to continuation c." This is the continuation `(cont` x_a `(call` x_f x_a *cont*`))` we see in the third case of the Fischer/Reynolds algorithm, represented as $\mathit{ACont}(x_f, \mathit{cont})$.

- *ICont*(e_1, e_2, E, c)
  This is the continuation that is awaiting the result of evaluating the "test" sub-expression of an `if` form. Its mission is: "If the value we receive is true, do the e_1 computation, delivering its value to continuation c; if it is false, do the e_2 computation instead." This is the continuation `(cont` x_b `...)` that appears in the fourth case of the Fischer/Reynolds algorithm.
  Again, note that this abstract continuation is constructed from source terms e_1 and e_2, so we also need a static environment E to assist the CPS translation of these two terms, when we decide to render this continuation into an actual CPS term.

Analogous to abstract continuations, there are two kinds of abstract arguments. The first kind is a variable x, from the CPS target language. The second kind $\langle[\![$`(fun` y e`)`$]\!], E\rangle$ represents the CPS `(lam (`x k`) ...)` term we will get when we translate source-language term `(fun` y e`)`. Because we are representing a CPS term with its direct-style source, we must package that source term up with a static environment E to tell us how to translate the free variables of the source term.

The point of introducing these abstract syntactic domains is that they will let the (eventual, smart) algorithm delay rendering their elements. As we'll see, the smart algorithm will have the flexibility to decide, based on context, to render an abstract element into its concrete syntax, or instead to reduce it away.

The final change we make to produce the Dumb algorithm from the Fischer/Reynolds algorithm is that we must render our abstract elements into the concrete syntactic terms they represent. We call this "blessing" an abstract-syntax value. Given some abstract value a, we write its correspondingly blessed concrete syntax term $\overline{a}$.

5.2 Syntax Constructors and the Dumb Algorithm

Figure 5 shows the Dumb algorithm. The central, top-level function is C_d. It has the same four cases as the F function, but with some differences:

- It now takes an extra static-environment argument E; when we call the function on the top-level source term, we pass in an empty initial environment, which is extended as the algorithm recurs down into nested `fun` source terms.
- Instead of directly constructing continuation terms, it creates abstract continuations using the *FCont*, *ACont*, and *ICont* constructors, *e.g.*, in the third and fourth cases of the function. Likewise, the new algorithm creates a static closure $\langle e, E\rangle$ instead of a concrete `lam` term, in the second case of the function.
- As the converter now traffics in abstract syntactic items, they must be rendered into concrete syntax by the two bless functions: one for continuations; the other, for arguments.
- Where the F function directly produces `ret`, `call` and `if` terms, this algorithm instead uses "constructor" functions Ret_d, $Call_d$, and If_d; they are trivial helper functions.

Inspection of the Dumb algorithm will show that inlining the auxiliary functions into the top-level C_d function will produce the Fischer/Reynolds algorithm—a minor variant that uses a symbol table to distinguish source variables from target variables.

Note that blessing abstract syntactic elements kicks off recursive bouts of CPS translation, as the various source-language elements packaged up by the abstract constructors must be translated into their final form. For example, blessing the function-value-awaiting continuation *FCont*(e_a, E, c') involves using C_d recursively to convert the argument term e_a. We can also see where the symbol table is extended: when blessing a static closure, as the algorithm prepares recursively to convert the body e of the closure's `(fun` y e`)` term.

$$C_d\ e\ E\ c \triangleq \begin{cases} Ret_d\ c\ (Ee) & e \in \text{DSVAR} \\ Ret_d\ c\ \langle e, E\rangle & e \in \text{FUN} \\ C_d\ e_f\ E\ FCont(e_a, E, c) & e = [\![\texttt{(}e_f\ \ e_a\texttt{)}]\!] \\ C_d\ e_1\ E\ ICont(e_2, e_3, E, c) & e = [\![\texttt{(if}\ e_1\ \ e_2\ \ e_3\texttt{)}]\!] \end{cases}$$

$$\begin{aligned} Ret_d\ c\ a &\triangleq [\![\texttt{(ret}\ \overline{c}\ \ \overline{a}\texttt{)}]\!] \\ Call_d\ f\ a\ c &\triangleq [\![\texttt{(call}\ \overline{f}\ \ \overline{a}\ \ \overline{c}\texttt{)}]\!] \\ If_d\ a\ e_1\ e_2\ E\ c &\triangleq [\![\texttt{(letc (}j\ \overline{c}\texttt{) (if}\ \overline{a}\ (C_d\ e_1\ E\ j)\ (C_d\ e_2\ E\ j)\texttt{)))}]\!] \end{aligned}$$

$$\overline{c} \triangleq \begin{cases} c & c = [\![\texttt{halt}]\!] \lor c \in \text{CVAR} \\ [\![\texttt{(cont}\ x_f\ \ (C_d\ e_a\ E\ ACont(x_f, c'))\texttt{)}]\!] & c = FCont(e_a, E, c') \\ [\![\texttt{(cont}\ x_a\ \ (Call_d\ a_f\ x_a\ c')\texttt{)}]\!] & c = ACont(a_f, c') \\ [\![\texttt{(cont}\ x\ \ (If_d\ x\ e_1\ e_2\ E\ c')\texttt{)}]\!] & c = ICont(e_1, e_2, E, c') \end{cases}$$

$$\overline{a} \triangleq \begin{cases} a & a \in \text{UVAR} \\ [\![\texttt{(lam (}x\ \ k\texttt{)}\ \ (C_d\ e\ E[y \mapsto x]\ k)\texttt{)}]\!] & a = \langle [\![\texttt{(fun}\ \ y\ \ e\texttt{)}]\!], E\rangle \end{cases}$$

Fig. 5. The dumb algorithm

5.3 The Smart Algorithm

The smart algorithm, shown in Figure 6, is a natural extension of the dumb one: we simply exploit the machinery we've put into place. Our first extension is to enrich the domain of the symbol table E, so that we can bind source-language variables either to target-language variables or to static closures. This is the key step that permits us to perform reductions on the fly: to substitute a term a for a variable x, we simply enter the mapping in the symbol table and proceed with the CPS conversion. We write the smart bless functions as $\overrightarrow{c}$ and $\overrightarrow{a}$, to visually distinguish them from their dumb counterparts.

Secondly, while the top-level function remains the same, our helper functions are now "smart" constructors. That is, $Ret\ c\ a$ doesn't simply produce the CPS term returning a to continuation c, that is, `(ret` $\overrightarrow{c}\ \overrightarrow{a}$`)`. Instead, it produces the reduced, No-Brainer normal form of that term. Likewise for the other helper-function constructors.

One such reduction opportunity arises when the $Call$ function is constructing a `call` term whose function part f is a static closure $\langle [\![\texttt{(fun}\ \ y\ \ e\texttt{)}]\!], E\rangle$. In this case, the `call` term is a β-redex. If y is a single-reference variable, then we have a $\beta_{\lambda 1}$ opportunity, so we don't want to produce the redex. Instead, we make a $y \mapsto a$ entry in the symbol table and CPS convert the body e of the function.

On the other hand, if y is multiply referenced in the body of the `fun`, we might still have a β_{cv} opportunity, if the substituend a will be rendered by the converter as a (replicatable) CPS variable. Note that a could be something complex, that is, a static closure over a large `fun` term, and still get rendered as a single CPS variable, if the result `lam` term η-reduces to a single variable. For example, if $a = \langle [\![\texttt{(fun q (((fun s s) r) q))}]\!], E\rangle$, it will be blessed to its No-Brainer normal form `r` (well, to its equivalent CPS variable, that is), in which case the algorithm will be able to use it as the substituend in a β_{cv} reduction. Again, this reduction is performed simply by making a $[y \mapsto triv]$ addition to the symbol table, where $triv$ is the CPS variable produced by blessing a, and then recursively translating the body of the function.

$$C\ e\ E\ c \triangleq \begin{cases} \mathit{Ret}\ c\ (Ee) & e \in \text{DSVAR} \\ \mathit{Ret}\ c\ \langle e, E\rangle & e \in \text{FUN} \\ C\ e_f\ E\ \mathit{FCont}(e_a, E, c) & e = \llbracket \texttt{(}e_f\ \ e_a\texttt{)} \rrbracket \\ C\ e_1\ E\ \mathit{ICont}(e_2, e_3, E, c) & e = \llbracket \texttt{(if}\ \ e_1\ \ e_2\ \ e_3\texttt{)} \rrbracket \end{cases}$$

$$\mathit{Ret}\ c\ a \triangleq \begin{cases} \llbracket \texttt{(ret}\ \ c\ \ \overrightarrow{a}\texttt{)} \rrbracket & c \in \text{CVAR} \lor c = \llbracket \texttt{halt} \rrbracket \\ C\ e\ E\ \mathit{ACont}(a, c') & c = \mathit{FCont}(e, E, c') \\ \mathit{Call}\ f\ a\ c' & c = \mathit{ACont}(f, c') \\ \mathit{If}\ a\ e_1\ e_2\ E\ c' & c = \mathit{ICont}(e_1, e_2, c', E) \end{cases}$$

$$\mathit{Call}\ f\ a\ c \triangleq \begin{cases} \llbracket \texttt{(call}\ \ f\ \ \overrightarrow{a}\ \ \overrightarrow{c}\texttt{)} \rrbracket & f \in \text{UVAR} \\ \text{if } \mathit{nref}(y, e) = 1 \text{ then } C\ e\ E[y \mapsto a]\ c & f = \langle \llbracket \texttt{(fun}\ \ y\ \ e\texttt{)} \rrbracket, E\rangle \\ \text{else let } \mathit{triv} = \overrightarrow{a} & \\ \quad \text{if } \mathit{triv} \in \text{UVAR then } C\ e\ E[y \mapsto \mathit{triv}]\ c & \\ \quad \text{else let } \mathit{body} = C\ e\ E[y \mapsto x]\ c & \\ \qquad \llbracket \texttt{(ret}\ \ \texttt{(cont}\ \ x\ \ \mathit{body}\texttt{)}\ \ \mathit{triv}\texttt{)} \rrbracket & \end{cases}$$

$$\mathit{If}\ a\ e_1\ e_2\ E\ c \triangleq \begin{cases} \llbracket \texttt{(if}\ \ \overrightarrow{a}\ \ (C\ e_1\ E\ c)\ \ (C\ e_2\ E\ c)\texttt{)} \rrbracket & c = \llbracket \texttt{halt} \rrbracket\ \lor\ c \in \text{CVAR} \\ \llbracket \texttt{(letc}\ \ \texttt{(}j\ \ \overrightarrow{c}\texttt{)}\ \ (\mathit{If}\ a\ e_1\ e_2\ E\ j)\texttt{)} \rrbracket & \mathit{otherwise} \end{cases}$$

$$\overrightarrow{c} \triangleq \begin{cases} c & c = \llbracket \texttt{halt} \rrbracket\ \lor\ c \in \text{CVAR} \\ \llbracket \texttt{(cont}\ \ x\ \ (\mathit{Ret}\ c\ x)\texttt{)} \rrbracket & \mathit{otherwise} \end{cases}$$

$$\overrightarrow{a} \triangleq \begin{cases} a & a \in \text{UVAR} \\ \text{let } b = C\ e\ E[y \mapsto x]\ k & a = \langle \llbracket \texttt{(fun}\ \ y\ \ e\texttt{)} \rrbracket, E\rangle \\ \text{if } \mathit{nref}(x, b) = 1 \text{ and } b = \llbracket \texttt{(call}\ \ \mathit{triv}\ \ x\ \ k\texttt{)} \rrbracket & \\ \text{then } \mathit{triv} \quad \textit{(* } \eta\textit{-reduction *)} & \\ \text{else } \llbracket \texttt{(lam}\ \ \texttt{(}x\ \ k\texttt{)}\ \ b\texttt{)} \rrbracket & \end{cases}$$

Fig. 6. The smart algorithm

The last two lines of the *Call* function handle the case `((fun` y e`) (fun ...))` where y's multiple references in e block No-Brainer reduction. This is rendered with a "let" binding encoded as a `ret`/`cont` redex: `(ret (cont` x *body*`) (lam (`x' k'`) ...))`, where x is the CPS variable chosen for source-variable y, *body* is the result of CPS-converting e, and `(lam (`x' k'`) ...)` is the CPS rendering of the un-substitutable `(fun ...)` term.

The final reduction opportunity comes when we bless a static closure $\langle[\![$`(fun` y e`)`$]\!], E\rangle$. This is where we have the possibility of an η-reduction. These are detected by first CPS converting the body e of the `fun`, and then checking to see if the result body has the form we require of an η-redex, that is, `(call` *triv* x k`)`, where x and k are the two fresh CPS parameters introduced for the `lam` term we are constructing. If that call's reference to x is the only reference x has, then instead of producing the term `(lam (`x k`) (call` *triv* x k`))`, we simply produce its contraction *triv*. We do not need to check to see if the k parameter occurs in *triv*, because continuation variables obey a linear discipline [Sabry and Felleisen 1993]; thus, if k is used as the continuation parameter in `(call` *triv* x k`)`, we know it cannot also be referenced within *triv*. (However, if our source language included first-class control operators, such as `call/cc`, the linearity constraint would no longer hold, and the algorithm would need to count references to continuation variables as well as user variables.)

The η-reducer's $\mathit{nref}(x, b) = 1$ check means that the smart algorithm needs to keep track of variable-reference counts in the target term as it is constructed. To keep the presentation simple, we have not shown this machinery in the algorithm, but it is straightforward: the various recursive functions pass around a second symbol table, one that maps CPS variables to numeric reference counts. Recurring into the new scope of a `lam` term adds a new $[x \mapsto 0]$ entry to this table; when we bless an abstract argument and produce a variable reference x, we increment its reference count in the table. (The implementation in the appendix includes this code.)

The key to the smart algorithm is our use of abstract continuations and arguments to delay the render-or-reduce decision until a value has flowed to its final resting place. If that place occurs in a No-Brainer redex, we can do the reduction on-the-fly simply by adding an entry to our symbol table. The symbol table is the key enabler of the fused conversion/normalising algorithm, in that it is the means by which we do reductions as we CPS convert.

Besides the two uses of a symbol table to (1) perform reductions and (2) track variable reference counts in the target term, there is a third use of a symbol table that we've glossed over due to its simplicity. Before CPS converting a source term, we must first walk the term counting variable references for each source variable, marking the singly-referenced variables as candidates for $\beta_{\lambda 1}$ reductions. This is a simple, linear-time pre-pass. Thus, the entire CPS conversion requires two, linear-time tree-walks of the source term.

6 PROOFS OF PROPERTIES

Up to this point, we have asserted that the No-Brainer reduction system and our algorithm have various favorable properties. In this section, we formally support those claims.

6.1 Properties of NBR

No-Brainer reduction is designed to be a greedy optimizer of term size, which we inductively define for our CPS language in Figure 7. Note that the size of a `lam` term is defined to agree with its corresponding λ-calculus equivalent, which is made of a curried λ-term.

The following proofs are modeled after those in Barendregt's standard text [1985]. Similar modified versions of the Barendregt proofs can be found in Appel and Jim's "Shrinking lambda expressions in linear time" [1997].

THEOREM 6.1 (TERM-SIZE REDUCTION).

$$\forall p_1, p_2 \in CPS,\ p_1 \xrightarrow[NB]{} p_2 \implies \mathit{size}(p_1) > \mathit{size}(p_2).$$

Proof Sketch: By case analysis on the definition of NBR, where size is defined as in Figure 7. □

$$\begin{aligned}
size(x) &\triangleq 1\\
size[\![\texttt{(lam (}x\ k\texttt{)}\ p\texttt{)}]\!] &\triangleq size(p) + 2\\
size[\![\texttt{(call}\ t_1\ t_2\ c\texttt{)}]\!] &\triangleq size(t_1) + size(t_2) + size(c) + 1\\
size[\![\texttt{(ret}\ c\ t\texttt{)}]\!] &\triangleq size(c) + size(t) + 1\\
size[\![\texttt{(if}\ t\ p_1\ p_2\texttt{)}]\!] &\triangleq size(t) + size(p_1) + size(p_2) + 1\\
size[\![\texttt{(letc (}k\ c\texttt{)}\ p\texttt{)}]\!] &\triangleq size(c) + size(p) + 1\\
size(k) &\triangleq 1\\
size[\![\texttt{(cont}\ x\ p\texttt{)}]\!] &\triangleq size(p) + 1\\
size[\![\texttt{halt}]\!] &\triangleq 1
\end{aligned}$$

Fig. 7. The size of a CPS term

This property is essential to the notion of No-Brainer reductions. The reductions are "no-brainer" precisely because they cause an immediate and obvious reduction in term size while preserving semantics.

Theorem 6.2 (Strong Normalisation).

$$\forall p \in CPS,\ \nexists\ \textit{an infinite reduction sequence}\ p \underset{NB}{\longrightarrow} p' \underset{NB}{\longrightarrow} p'' \underset{NB}{\longrightarrow} \dots$$

Proof Sketch: By Theorem 6.1 and structural induction on the term, using term size as a measure to ensure termination. □

We now have a guarantee that there are no infinite reduction paths in NBR. Thus, we can implement the system algorithmically without having to worry whether the optimisations will fail to terminate. If this were not the case, attempts to reduce a term might never reach a normal form.

While classic $\beta\eta$-reduction is known to be confluent, we must prove this property for NBR. We will accomplish this by proving each individual reduction of the system is confluent, and then demonstrating that they commute.

Lemma 6.3 (Local Confluence of β_{cv} and $\beta_{\lambda 1}$). $\forall p \in CPS,$

$$\begin{tikzcd}
p \arrow[r, "\beta_{cv}"] \arrow[d, "\beta_{cv}"'] & p' \arrow[d, dashed, "\beta^*_{cv}"] \\
p'' \arrow[r, dashed, "\beta^*_{cv}"] & p'''
\end{tikzcd}
\qquad
\begin{tikzcd}
p \arrow[r, "\beta_{\lambda 1}"] \arrow[d, "\beta_{\lambda 1}"'] & p' \arrow[d, dashed, "\beta^*_{\lambda 1}"] \\
p'' \arrow[r, dashed, "\beta^*_{\lambda 1}"] & p'''
\end{tikzcd}$$

Proof Sketch: For each reduction, we inductively define a relation with the intention that the reduction is its transitive closure. So we prove both β_{cv} and $\beta_{\lambda 1}$ are locally confluent by proving that the corresponding relations are both locally confluent. □

LEMMA 6.4 (COMMUTATIVITY OF β_{cv} AND $\beta_{\lambda 1}$). $\forall p \in CPS$,

$$\begin{tikzcd}
p \arrow[r, "\beta_{cv}"] \arrow[d, "\beta_{\lambda 1}"] & p' \arrow[d, dashed, "\beta^*_{\lambda 1}"] \\
p'' \arrow[r, dashed, "\beta^*_{cv}"] & p'''
\end{tikzcd}$$

Proof Sketch: By case analysis on the definition of β_{cv}. We accomplish this using context grammars, considering a particular β_{cv} redex within a term. We present three of these cases diagrammatically in Figure 8; the remaining three cases are similarly structured. □

For the remainder of the section, we use β to refer to β_{cv} and $\beta_{\lambda 1}$ combined.

LEMMA 6.5 (CONFLUENCE OF NO-BRAINER β-REDUCTION). $\forall p \in CPS$,

$$\begin{tikzcd}
p \arrow[r, "\beta"] \arrow[d, "\beta"] & p' \arrow[d, dashed, "\beta^*"] \\
p'' \arrow[r, dashed, "\beta^*"] & p'''
\end{tikzcd}$$

Proof Sketch: By Lemma 6.3, Lemma 6.4, and Proposition 3.3.5 in Barendregt [1985]. □

Classical η-reduction is known to be confluent [Barendregt 1985], so to show that the No-Brainer reduction system is also confluent, it suffices to show that η-reduction commutes with our restricted β-reduction.

LEMMA 6.6 (COMMUTATIVITY OF β AND η). $\forall p \in CPS$,

$$\begin{tikzcd}
p \arrow[r, "\beta"] \arrow[d, "\eta"] & p' \arrow[d, dashed, "\eta^*"] \\
p'' \arrow[r, dashed, "\beta^*"] & p'''
\end{tikzcd}$$

Proof Sketch: Diagrammatically using reduction context, as in our proof of Lemma 6.4. We consider every possible η-redex, performing β-reduction in context and subexpressions. □

THEOREM 6.7 (GLOBAL CONFLUENCE OF NBR). $\forall p \in CPS$,

$$\begin{tikzcd}
p \arrow[r, "NBR"] \arrow[d, "NBR"] & p' \arrow[d, dashed, "NBR^*"] \\
p'' \arrow[r, dashed, "NBR^*"] & p'''
\end{tikzcd}$$

Proof Sketch: By Lemma 6.5, confluence of η-reduction, and Lemma 6.6. □

$$\begin{tikzcd}
 & & C[\texttt{(call (lam (}x_1\ k\texttt{)}\ p\texttt{)}\ x_2\ \mathit{cont}\texttt{)}] \arrow[dll, "\beta_{\lambda 1}"'] \arrow[ddl, "\beta_{\lambda 1}"] \arrow[ddd, "\beta_{cv}"] \arrow[ddr, "\beta_{\lambda 1}"'] \arrow[drr, "\beta_{\lambda 1}"] & & \\
C'\left[\begin{array}{l}\texttt{(call (lam (}x_1\ k\texttt{)}\ p\texttt{)}\\ \quad x_2\\ \quad \mathit{cont}\texttt{)}\end{array}\right] \arrow[ddd, dashed, "\beta_{cv}"'] & & & & C\left[\begin{array}{l}\texttt{(ret (cont}\ x_1\ p[k\mapsto \mathit{cont}]\texttt{)}\\ \quad x_2\texttt{)}\end{array}\right] \arrow[ddd, dashed, "\beta_{cv}"] \\
 & C\left[\begin{array}{l}\texttt{(call (lam (}x_1\ k\texttt{)}\ p'\texttt{)}\\ \quad x_2\\ \quad \mathit{cont}\texttt{)}\end{array}\right] \arrow[ddd, dashed, "\beta_{cv}"] & & C\left[\begin{array}{l}\texttt{(call (lam (}x_1\ k\texttt{)}\ p\texttt{)}\\ \quad x_2\\ \quad \mathit{cont}'\texttt{)}\end{array}\right] \arrow[ddd, dashed, "\beta_{cv}"'] & \\
 & & C\left[\begin{array}{l}\texttt{(letc (}k\ \mathit{cont}\texttt{)}\\ \quad p[x_1\mapsto x_2]\texttt{)}\end{array}\right] \arrow[dll, dashed, "\beta_{\lambda 1}"'] \arrow[ddl, dashed, "\beta_{\lambda 1}"'] \arrow[ddr, dashed, "\beta_{\lambda 1}"] \arrow[drr, dashed, "\beta_{\lambda 1}"] & & \\
C'\left[\begin{array}{l}\texttt{(letc (}k\ \mathit{cont}\texttt{)}\\ \quad p[x_1\mapsto x_2]\texttt{)}\end{array}\right] & & & & C[p[k\mapsto \mathit{cont}][x_1\mapsto x_2]] \\
 & C\left[\begin{array}{l}\texttt{(letc (}k\ \mathit{cont}\texttt{)}\\ \quad p'[x_1\mapsto x_2]\texttt{)}\end{array}\right] & & C\left[\begin{array}{l}\texttt{(letc (}k\ \mathit{cont}'\texttt{)}\\ \quad p[x_1\mapsto x_2]\texttt{)}\end{array}\right] &
\end{tikzcd}$$

(a) Commutativity of $\beta_{\lambda 1}$ with the first β_{cv} rule

$$\begin{tikzcd}
 & & C[\texttt{(ret (cont}\ x_1\ p\texttt{)}\ x_2\texttt{)}] \arrow[dll, "\beta_{\lambda 1}"'] \arrow[dd, "\beta_{cv}"] \arrow[drr, "\beta_{\lambda 1}"] & & \\
C'[\texttt{(ret (cont}\ x_1\ p\texttt{)}\ x_2\texttt{)}] \arrow[ddr, dashed, "\beta_{cv}"'] & & & & C[\texttt{(ret (cont}\ x_1\ p'\texttt{)}\ x_2\texttt{)}] \arrow[ddl, dashed, "\beta_{cv}"] \\
 & & C[p[x_1\mapsto x_2]] \arrow[dl, dashed, "\beta_{\lambda 1}"'] \arrow[dr, dashed, "\beta_{\lambda 1}"] & & \\
 & C'[p[x_1\mapsto x_2]] & & C[p'[x_1\mapsto x_2]] &
\end{tikzcd}$$

(b) Commutativity of $\beta_{\lambda 1}$ with the fourth β_{cv} rule

$$\begin{tikzcd}
 & & C[\texttt{(letc (}k_1\ k_2\texttt{)}\ p\texttt{)}] \arrow[dll, "\beta_{\lambda 1}"'] \arrow[dd, "\beta_{cv}"] \arrow[drr, "\beta_{\lambda 1}"] & & \\
C'[\texttt{(letc (}k_1\ k_2\texttt{)}\ p\texttt{)}] \arrow[ddr, dashed, "\beta_{cv}"'] & & & & C[\texttt{(letc (}k_1\ k_2\texttt{)}\ p'\texttt{)}] \arrow[ddl, dashed, "\beta_{cv}"] \\
 & & C[p[k_1\mapsto k_2]] \arrow[dl, dashed, "\beta_{\lambda 1}"'] \arrow[dr, dashed, "\beta_{\lambda 1}"] & & \\
 & C'[p[k_1\mapsto k_2]] & & C[p'[k_1\mapsto k_2]] &
\end{tikzcd}$$

(c) Commutativity of $\beta_{\lambda 1}$ with the fifth β_{cv} rule

Fig. 8. Commutativity of β_{cv} and $\beta_{\lambda 1}$

Now that we have proven the No-Brainer system normalising and confluent, we can refer to "the normal form" of a given term and be guaranteed that it is unique, no matter which reductions we take locally. This allows us to implement our CPS-conversion algorithm deterministically, as we can make reductions everywhere we find a redex and not worry whether it will lead to a different translated term.

6.2 Properties of the Algorithm

For our algorithm to match our specification, we must prove two key properties: that it preserves the semantics of the original term and that its output is in No-Brainer normal form.

One issue we must address first, however, is the fact that the environment argument passed around by the converter will always be sufficient to the needs of the variable lookups performed on it. This is a little subtle, because an environment contains other environments, packaged up inside the static closures in the first environment's range. Ensuring the algorithm always "hits" when it indexes into an environment on some lookup amounts to a precondition on the environment argument passed to any of the converter functions, one which is preserved as environments are manipulated during the execution of the converter.

We define a well-formedness property W for term-environment pairs and an analogous property WC for continuations, so that when we convert a term with this property, variable lookups will always be well defined. Theorem 6.10 guarantees that this property is preserved on all recursions of all algorithms we define that use the environment.

Definition 6.8 (Environment well-formedness property). An environment E is well-formed with respect to a direct-style term e (written $\langle e, E\rangle \in W$) if every free variable in e is in the domain of the environment, and every static closure $\langle lam, E'\rangle$ in the range of E is itself a well-formed term/environment pair:

$$\langle e, E\rangle \in \mathrm{W} \text{ iff } FV(e) \subset domain(E) \ \wedge \ range(E) - \mathrm{UVAR} \subset \mathrm{W}.$$

Definition 6.9 (Well-formed continuation). An abstract continuation c is well formed (written $c \in \mathrm{WC}$) if all environments packaged up inside the continuation are well formed with respect to their associated syntax elements, and all abstract continuations within c are also well formed:

$$c \in \mathrm{WC} \text{ iff } \begin{cases} \text{true} & c \in \mathrm{CVAR} \\ \langle e, E\rangle \in \mathrm{W} \ \wedge c \in \mathrm{WC} & c = FCont(e, E, c) \\ (a \in \mathrm{W} \vee a \in \mathrm{UVAR}) \ \wedge \ c \in \mathrm{WC} & c = ACont(a, c) \\ \langle e_1, E\rangle, \langle e_2, E\rangle \in \mathrm{W} \ \wedge \ c \in \mathrm{WC} & c = ICont(e_1, e_2, E, c). \end{cases}$$

THEOREM 6.10 (PRESERVATION OF WELL-FORMEDNESS).

(1) *C_d e E c preserves $\langle e, E\rangle \in W$ and $c \in WC$ on all recursions.*
(2) *C e E c preserves $\langle e, E\rangle \in W$ and $c \in WC$ on all recursions.*

Proof Sketch: Induction on the structure of the term and mutual induction on the various smart constructors. □

The next two lemmas are necessary for us to prove that the Smart algorithm produces a No-Brainer reduction of the Dumb algorithm's output. Our goal is to formalise the notion that when we recur with an extended environment, we are β-reducing the term. To prove this equivalence, we must extend variable substitution to operate on unblessed, abstract continuations (Figure 9), and prove its equivalence to substitution on concrete, syntactic continuations in our first lemma.

$$\begin{aligned}
\texttt{halt}[k \mapsto c] &\triangleq \texttt{halt} \\
k'[k \mapsto c] &\triangleq \begin{cases} c & k = k' \\ k' & k \neq k' \end{cases} \\
\mathit{ACont}(a, c')[k \mapsto c] &\triangleq \mathit{ACont}(a, c'[k \mapsto c]) \\
\mathit{FCont}(e, E, c')[k \mapsto c] &\triangleq \mathit{FCont}(e, E, c'[k \mapsto c]) \\
\mathit{ICont}(e_1, e_2, E, c')[k \mapsto c] &\triangleq \mathit{ICont}(e_1, e_2, E, c'[k \mapsto c])
\end{aligned}$$

Fig. 9. The substitution $[k \mapsto c]$ is extended to operate on the elements of the abstract ABS-CONT domain, respecting their interpretation as concrete CPS terms.

LEMMA 6.11 (ABSTRACT CONTINUATION SUBSTITUTION). *Substitution on an abstract continuation respects its reification as concrete syntax:*

$\forall c, c', k \in WC,\ \forall \langle e, E\rangle \in W,$

(1) $\overline{c}[k \mapsto \overline{c'}] = \overline{c[k \mapsto c']}$, *and*

(2) $C_d\ e\ E\ c[k \mapsto c'] = \big(C_d\ e\ E\ c\big)[k \mapsto \overline{c'}]$.

Proof Sketch: By mutual induction across C_d and the blessing functions, the structure of the continuation, and the structure of the e term. The proof of the first half of the theorem stands alone through the k and $\mathit{ACont}(a, c)$ cases, but the $\mathit{FCont}(e, E, c)$ case requires the algorithm to step through C_d. To solve this issue, we add the second half of this theorem which is proven by induction on the structure of the term and continuation. This property allows us to use our inductive hypothesis to move nested continuations out of the data structure, reify them, and perform the outer substitution before reversing the process to recreate the data structure post-substitution. □

Now that we have a well-understood definition of continuation substitution, we can prove that β-reduction is equivalent to environment extension.

LEMMA 6.12 (ENVIRONMENT EXTENSION IS β-REDUCTION).

$\forall \langle e, E[y \mapsto a]\rangle \in W,\ \forall c \in WC,$

$C_d\ e\ E[y \mapsto a]\ c = \big(C_d\ e\ E[y \mapsto x]\ c\big)[x \mapsto \overline{a}]$

where x is fresh.

Proof Sketch: By induction on the structure of the direct-style term with additional cases for variables not equal to y. We use the continuation substitution property established in Lemma 6.11 to substitute into the continuation data structure. This lets us perform substitutions on continuations by partially reifying them. First, we replace a continuation data structure with a continuation variable and a substitution. Then we will swap the inner and outer substitutions using the fact that parallel λ-calculus substitutions compose. Finally, we substitute into the blessed continuation and then convert it back into a data structure by running the continuation-blessing operation $\overline{c}$ backwards. This allows us to propagate environment extensions into continuation data structures. □

With these lemmas in hand, we can prove that the Smart algorithm produces the NBNF of the Dumb algorithm's output.

THEOREM 6.13 (DUMB ALGORITHM $\xrightarrow[NB]{}^*$ SMART ALGORITHM). *The output of the Dumb algorithm reduces to the output of the Smart algorithm:*

$\forall \langle e, E\rangle, \langle e_1, E\rangle, \langle e_2, E\rangle \in W,\ \forall a, a' \in W,\ \forall c \in WC:$

$$C_d\ e\ E\ c \xrightarrow[NB]{}^* C\ e\ E\ c,$$
$$Ret_d\ c\ a \xrightarrow[NB]{}^* Ret\ c\ a,$$
$$Call_d\ a\ a'\ c \xrightarrow[NB]{}^* Call\ a\ a'\ c,$$
$$If_d\ a\ e_1\ e_2\ E\ c \xrightarrow[NB]{}^* If\ a\ e_1\ e_2\ E\ c,$$
$$\overline{a} \xrightarrow[NB]{}^* \overrightarrow{a}, \text{ and}$$
$$\overline{c} \xrightarrow[NB]{}^* \overrightarrow{c}.$$

Proof Sketch: By double induction on the structure of the source term and continuation data structure and mutual induction between the various smart constructors. Lemma 6.12 is used to relate the two algorithms through β-reduction. □

As a consequence, we can conclude that converting a top-level, closed term with the Smart algorithm produces a CPS term that is just a simplified version of the output produced by the simple Dumb algorithm:

COROLLARY 6.14. $FV(e) = \varnothing \implies C_d\ e\ [\cdot]\ \texttt{halt} \xrightarrow[NB]{}^* C\ e\ [\cdot]\ \texttt{halt}.$

So we now know the Smart algorithm is *correct*, in that it respects the Dumb algorithm; and it might even produce a term that is simpler. But is it *optimal*? That is, does it produce a No-Brainer normal form? This is established with the following theorem.

THEOREM 6.15 (THE SMART-ALGORITHM OUTPUT IS IN NBNF). $\forall a \in W,\ \forall c \in WC,$ $\forall \langle e, E\rangle, \langle e_1, E\rangle, \langle e_2, E\rangle \in W$, *the following terms are in No-Brainer normal form:*

- $C\ e\ E\ c$
- $Ret\ c\ a$
- $Call\ a_1\ a_2\ c$
- $If\ a\ e_1\ e_2\ E\ c$
- $\overrightarrow{a}$
- $\overrightarrow{c}$

Proof Sketch: As Lemma 6.12 formalises the notion that environment extension is equivalent to β-reduction, proving that we generate a No-Brainer normal form is simply a matter of verifying that we extend the environment in the correct places. We also need to consider the η-reduction case, which can be done by inspecting the argument-blessing function $\overrightarrow{a}$. Finally, we must ensure that we have reduced continuations whenever possible. This requires us to examine the cases of *Ret*, which shows that we do indeed delegate to smart constructors that perform these reductions where appropriate. □

With slight modifications, Theorem 6.12 and Theorem 6.15 can be combined by verifying that the reductions that transform the Dumb algorithm into the Smart algorithm are the No-Brainer reductions and that the Smart algorithm is in NBNF. We separate these results into two theorems for clarity.

Finally, we should consider the time complexity of the algorithm. The algorithm makes two passes over the input, once to compute variable reference counts in the source, and once to do the actual translation to CPS. The time complexity, then, is determined by the cost of the symbol-table operations. If we assume they are $O(\log n)$ (*e.g.*, red-black trees), then the total cost is $O(n \log n)$; if we assume they are constant-time (*e.g.*, imperative hash tables), then the total cost is $O(n)$.

7 VARIATIONS

We've focussed the development of our algorithm, so far, on a core λ-calculus: variables, λ-terms, applications and a primitive conditional. But once the central ideas of the algorithm are understood, multiple variations on the basic theme are possible.

7.1 Other Reductions

We can easily extend the algorithm to handle other kinds of simplifying, "no-brainer" local reductions at translation time. For example, if we extend our source and CPS language to include literal constants other than `halt`, we can use our environments to perform constant propagation. This more or less comes for free by virtue of the fact that the algorithm is built around the use of a symbol table. Just as when substituting λ-terms, constant-propagating reductions can be disallowed when the constant substituend is large (*e.g.*, a list rather than an integer or boolean) and the parameter being reduced away has multiple references in the body of its binding term.

The counts we create in our first pass over the source tree could also be used for dead-variable elimination, though this can lead to new reduction opportunities which we would not capture in our current algorithm. This would mean that our term would not be in No-Brainer normal form, but would be strictly smaller while preserving semantics. Additionally, the occurrence counts generated during the conversion process can be used to perform another pass over the source tree to remove more, though not all, dead variables and further reduce away new No-Brainer redexes. This approach is almost identical to the one used in Appel and Jim's contract algorithm [1997].

We can also do constant folding, when known primitive functions are applied to constant arguments (perhaps by virtue of the constant propagation described above—these simplifications cascade). We can also fold away conditionals with known tests, *e.g.*, reducing `(if false` e_1 e_2`)` to e_2.

It's probably wise not to jam too much complexity into a CPS-converting front end, unless, perhaps, we were writing a fast, simple compiler that did no other optimisation at all. In a front-end for an optimising, multi-pass compiler, the point is to do, judiciously, the easy things—to clear away the "underbrush"[2] in a simple, linear-time way before proceeding to the more complex, costly transformation phases of a compiler.

7.2 First-class Control Operators

We can extend the algorithm to handle first-class control operators such as `call/cc` with three changes. First, we introduce two new abstract trivial arguments: the constant `call/cc`, and a constructor $\mathit{UCont}(c)$, which represents a continuation that has been captured by `call/cc` and exported to the source program as a user value. The *Call* function reduces applications Call `call/cc` $a\ c$ to $\mathit{Call}\ a\ \mathit{UCont}(c)\ c$, and *UCont* applications $\mathit{Call}\ \mathit{UCont}(c)\ a\ c'$ to $\mathit{Ret}\ c\ a$. Second, we license the β_{cv}-reduction code to replicate our two new abstract arguments. Third, since continuation variables no longer obey a linear discipline, we must extend the η-reduction guard also to check the reference count of the continuation parameter as well as the user-argument parameter.

7.3 Multiple Parameters and letc

In an implementation of our algorithm that is engineered for translating terms from a real programming language, we would likely extend λ-terms to permit both multiple user parameters and multiple continuation parameters. Even when translating languages such as SML, OCaml, or Haskell, where functions only take a single argument and return a single value, in the CPS intermediate representation we can usefully exploit multi-parameter functions and continuations to represent spreading values out in the register set across calls and returns, or to describe callee-saves register-management

[2] "Underbrush" being sort of the negative image of "low-hanging fruit."

policies [Appel 2006]. Likewise, multiple continuation parameters can be used to encode both a main return point and an alternate exception-handler exit, or to encode functions that can be called with multiple return points [Shivers and Fisher 2006].

This affects the algorithm in that we can now do β-reduction on a piecemeal, per-parameter basis—something that is *not* possible when we encode a multi-parameter function by, *e.g.*, currying `(fun (a b c) ...)` into `(fun (a) (fun (b) (fun (c) ...)))`. That is, if the first term occurs in a β-redex, we can "reach into" the middle of the parameter list and substitute away the `b` parameter, even if the first `a` parameter cannot be substituted. Extending our algorithm to work in this fashion is straightforward.

Once we admit multi-parameter λ-terms, we also get the ability to have λ-terms that take *no* user parameters. This means we no longer need the special `(letc (`*k c*`)` *body*`)` form to bind the join points required for translating conditionals. Instead, we can encode the binding with a redex that applies a λ-term that binds one continuation but no user parameters:

```
(call (lam (k)          ; 1. Bind join cont k;
        (if x (ret k 1) ; 2. do conditional,
              (ret k 2) ;    then jump to
      (cont (y) ...))   ; 3. ...this join point.
```

This is more elegant; we elected not to do things this way in our main development so that we could use a simpler language where a λ term always binds exactly one user parameter and one continuation parameter.

7.4 ANF

The basic ideas of the algorithm can easily be carried over to one that translates direct-style terms to Felleisen and Sabry's ANF [1993].

7.5 Metacontinuations

If we Church-encode the elements of the ABS-CONT set (that is, values constructed by *FCont*, *ACont*, *etc.*), then we can get an algorithm that uses the clever "metacontinuation" representation introduced by Danvy and Filinski [1992]. In fact, we did exactly this in the first version of our algorithm. We shifted to the first-order/defunctionalised variant we have shown in this paper for simplicity and clarity. In particular, it simplifies the inductive proof to realise the continuations as elements of an inductive type, rather than elements drawn from a space of functions.

Expressing the algorithm in a first-order language also means it can more easily be directly translated to a non-functional language, such as C, and also means that it can be directly expressed in ACL2 [Kaufmann et al. 1990] for purposes of verification.

7.6 Can the Algorithm Be "Calculated"?

Now that we understand the general idea of this CPS algorithm, could we start with a simple CPS converter and a separate, simple No-Brainer normalisation function, and then *derive* or *calculate* our interleaved algorithm from the serial composition of the two functions, in the style of Bird [Bird and Wadler 1992] or Danvy [Danvy and Filinski 1992, Section 2]? For example, Danvy and his students have applied the technique of "program calculation," exploiting control-flow analysis, defunctionalisation, refunctionalisation, and other simple transforms, to derive similar term-processing algorithms for the λ calculus, including CPS converters [Danvy 2008; Millikin 2005]. Such a derivation would comprise an independent and elegant explanation of the algorithm's correctness.

We leave this question as a (hopefully fun) puzzle for the reader.

8 CONCLUSION

Before we summarise the point of this pearl, let's first state what the point is *not*. The earlier algorithms of Sabry, Felleisen, Danvy and Filinski eliminate exactly the redexes needed to bring a machine executing the CPS result into lock step with a machine executing the direct-style source. This is a result of theoretical interest, establishing the foundations on which we stand when we work with CPS—but it does not capture the concerns of compiler writers.

A compiler writer has a more relaxed criterion for a CPS transform: an observational equivalence that permits the target term's computation to stutter a bit as the machine executes. Some bits of the computation can be discharged at compile time; we can reduce the interior of a λ term before it executes. So the classic "administrative" redexes that are of such interest to the theoretician have no special status in the eyes of the compiler.

Despite this, the Danvy/Filinski algorithm has become the standard algorithm of choice when language implementors need a front-end for a CPS-based compiler [Kennedy 2007]. This isn't because implementors want to produce a term that stays in exact lock-step with a simple machine operating on the source program. Implementors are attracted to this algorithm because all "administrative" redexes are No-Brainer redexes: there's no point in producing them if the compiler definitely wants to subsequently eliminate them.

The point, then, of this pearl is two-fold. First, we want to highlight the *idea* of the "No-Brainer" redex. It is similar to *but not the same as* an "administrative" redex, and it's the No-Brainer redex that's of interest to implementors.

Second, we want to draw the attention of implementors writing CPS and ANF front-ends to the lovely possibilities that occur when one simply *adds a symbol table* to the conversion algorithm. A symbol table, coupled with the idea of abstract closures and smart constructors, enables an algorithm that is short, simple and fast, yet it produces better results than the existing alternatives.

ACKNOWLEDGMENTS

Shivers worked out the initial version of the conversion algorithm during a very productive and enjoyable visiting appointment at the University of Aarhus in 2004, hosted by Olivier Danvy. There can hardly be a more pleasant intellectual experience than to discuss just about anything involving continuations and CPS with Danvy, whose encyclopedic knowledge of the topic is matched only by his irresistible enthusiasm for exploring its frontiers.

Earlier drafts of this paper received detailed, thoughtful and insightful reviews from the ICFP program committee that greatly improved the work and its presentation. Among other things, the entertaining question of Subsection 7.6 was posed by one of our reviewers. We are grateful for the time and care lavished on our submission by these anonymous reviewers.

Davis and Meehan received financial support from the Larry Finkelstein Innovation Fund that permitted them to work full time on this project over the summer of 2015. This fund was established at Northeastern's College of Computer Science by a gift from Brian Wenzinger, to enable undergraduate students to get research experience. All three authors would like to thank Wenzinger and the College of Computer Science for the opportunity this support made possible.

A OCAML IMPLEMENTATION

This implementation omits the trivial pre-pass that counts variable references to determine which source variables are single-reference variables. Instead, it assumes such variables have been marked with a colon prefix. Note that we could use a monad to hide the single-threading of the `count` table used to track the number of references made to user variables in the generated CPS terms. We've chosen to show this machinery to make it explicit.

```
type var = string

(* Create a unique string on each call. *)
let gensym =
  let counter = ref 0 in
  fun s -> (counter := 1 + !counter;
            Printf.sprintf "%s_%d" s !counter)

module SMap = Map.Make(String)  (* String -> alpha dictionary *)

(* Syntax of direct-style source language *)
type ds = Var of var
        | Fun of var * ds
        | App of ds * ds
        | DIf of ds * ds * ds

(* Syntax of CPS target language *)
type p = Call of triv * triv * cont
       | Ret  of cont * triv
       | CIf  of triv * p * p
       | Letc of var * cont * p
and cont = Cont of var * p
         | CVar of var
         | HALT
and triv = Lam  of var * var * p
         | UVar of var

(* Abstract args *)
type a = AVar of var                  (* A "user" var x, or          *)
       | AClo of var * ds * env       (* a <\y.e,env> static closure *)
and env = a SMap.t (* Env maps source var y to an abstract argument. *)

(* Abstract continuations *)
type c = AHALT
       | KVar  of var
       | FCont of ds * env * c
       | ACont of a * c
       | ICont of ds * ds * env * c

(* We assume singly-referenced vars are marked with a ":" prefix. *)
let one_ref y = (String.length y) >= 1 && (String.sub y 0 1) = ":"

(* Extend a static environment with a new [y |-> a] entry. *)
let extend y a env  = SMap.add y a env
```

```
(* Utilities to maintain reference counts of user vars in CPS term. *)
let new_count x counts = SMap.add x 0 counts
let incr      x counts = SMap.add x (1 + (SMap.find x counts)) counts

(* The top-level function *)
let rec cps exp env c counts =
  match exp with
  | Var y            -> ret c (SMap.find y env) counts
  | Fun (y, e)       -> ret c (AClo(y, e, env)) counts
  | App (e1, e2)     -> cps e1 env (FCont(e2, env, c))      counts
  | DIf (e1, e2, e3) -> cps e1 env (ICont (e2, e3, env, c)) counts

(* Three smart constructors, for RET, CALL & IF forms. *)

and ret c a counts =
  match c with
  | AHALT
  | KVar _ -> let (cont, counts2) = blessc c counts  in
              let (arg,  counts3) = blessa a counts2 in
              (Ret (cont, arg), counts3)
  | FCont(e, env, c')      -> cps e env (ACont(a, c')) counts
  | ACont(a', c')          -> call a' a c' counts
  | ICont(e1, e2, env, c') -> cif a e1 e2 c' env counts

and call f a c counts =
  match f with
  | AVar _ -> let (func, counts2) = blessa f counts  in
              let (arg,  counts3) = blessa a counts2 in
              let (cont, counts4) = blessc c counts3 in
              (Call(func, arg, cont), counts4)
  | AClo(y, body, env) ->
    if one_ref y then cps body (extend y a env) c counts
    else let (arg,  counts2) = blessa a counts in
         match arg with
         | UVar x -> cps body (extend y (AVar x) env) c counts2
         | Lam _  ->
           (* We've got a "let" redex, binding y to a lambda term:
            *     ((FUN y body) (FUN ...))
            * We can't reduce this because y has multiple references
            * in body, which would replicate the (FUN ...) term. So
            * we produce a CPS "let", encoded as a CONT redex:
            *     (RET (CONT x body') (LAM ...))
            * where body' is body cps-converted with the original
            * continuation c, and the (LAM ...) term is the
            * cps-conversion of the (FUN ...) argument.
            *)
           let x           = gensym "x"             in
           let counts3     = new_count x counts2    in
           let env'        = extend y (AVar x) env  in
           let (b,counts4) = cps body env' c counts3 in
           (Ret(Cont(x,b), arg), counts4)
```

```
and cif a e1 e2 c env counts =
  match c with
  | AHALT
  | KVar  _ -> let (test,   counts2) = blessa a counts      in
               let (conseq, counts3) = cps e1 env c counts2 in
               let (alt,    counts4) = cps e2 env c counts3 in
               (CIf(test, conseq, alt), counts4)
  | FCont _
  | ACont _
  | ICont _ ->
    let jv              = gensym "join"                   in
    let (body, counts2) = cif a e1 e2 (KVar jv) env counts in
    let (join, counts3) = blessc c counts2                in
    (Letc(jv, join, body), counts3)


(* Two "blessing" functions to render abstract continuations
   and abstract arguments into actual syntax. *)

and blessc c counts =
  match c with
  | AHALT   -> (HALT,    counts)
  | KVar kv -> (CVar kv, counts)
  | FCont _
  | ACont _
  | ICont _ -> let x               = gensym "x"            in
               let counts2         = new_count x counts    in
               let (body, counts3) = ret c (AVar x) counts2 in
               (Cont (x, body), counts3)

and blessa a counts =
  match a with
  | AVar x -> (UVar x, incr x counts)
  | AClo(y, body, env) ->
    let x    = gensym y              in
    let k    = gensym "k"            in
    let env' = extend y (AVar x) env in
    let (b, counts') = cps body env' (KVar k) (new_count x counts) in

    (* The eta-reduction check. Note that we don't have to check
       reference counts on k, as continuation variables are linear. *)
    match b with
    | Call(f, UVar x', CVar k') ->
      if x = x'  &&  k = k'  &&  (SMap.find x counts') = 1
      then (f,            counts')
      else (Lam (x, k, b), counts')
    | _ -> (Lam (x, k, b), counts')
```

REFERENCES

Andrew W. Appel. 2006. *Compiling with Continuations*. Cambridge University Press.

Andrew W. Appel and Trevor Jim. 1997. Shrinking lambda expressions in linear time. *Journal of Functional Programming* 7, 5 (1997), 515–540.

Hendrik Pieter Barendregt. 1985. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland, Amsterdam.

Richard Bird and Philip Wadler. 1992. *Introduction to Functional Programming*. Prentice Hall.

Olivier Danvy. 2008. Defunctionalized interpreters for programming languages. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP) (SIGPLAN Notices, Vol. 43, No. 9)*. 131–142. https://doi.org/10.1145/1411204.1411206

Olivier Danvy and Andrzej Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391. https://doi.org/10.1017/S0960129500001535

Michael J. Fischer. 1993. Lambda-calculus schemata. *LISP and Symbolic Computation* 6, 3/4 (1993), 259–288. Earlier version originally published in *Proceedings of the ACM Conference on Proving Assertions about Programs*, SIGPLAN Notices, Vol. 7, No. 1, and SIGACT News, No. 14, 104–109 (January 1972).

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI) (ACM SIGPLAN Notices, Vol. 28, No. 6)*. 237–247. https://doi.org/10.1145/155090.155113

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 1990. *Computer-Aided Reasoning: An Approach*. Number 3 in Advances in Formal Methods. Kluwer Academic Publishers.

Richard A. Kelsey. 1989. *Compilation by Program Transformation*. Ph.D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 702.

Richard A. Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Workshop on Intermediate Representations (SIGPLAN Notices, Vol. 30, No. 3)*. 13–22. https://doi.org/10.1145/202529.202532

Richard A. Kelsey and Paul Hudak. 1989. Realistic compilation by program transformation. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*. 281–292. https://doi.org/10.1145/75277.75302

Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP) (ACM SIGPLAN Notices, Vol. 42, No. 9)*. 177–190. https://doi.org/10.1145/1291151.1291179

David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. 1986. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction (SIGPLAN Notices, Vol. 21, No. 7)*. 219–233. https://doi.org/10.1145/12276.13333

David A. Kranz. 1988. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 632.

Kevin Millikin. 2005. A new approach to one-pass transformations. In *Trends in Functional Programming Volume 6*. Intellect Books, 252–264.

Gordon D. Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (Dec. 1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

Brian Rabern. 2016. The history of the use of $[\![\cdot]\!]$-notation in natural language semantics. *Semantics and Pragmatics* 9 (2016), 1–10. https://doi.org/10.3765/sp.9.12

John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th National ACM Conference*. 717–740. Reprinted in *LISP and Symbolic Computation*, 11 363–397 (1998).

John C. Reynolds. 1993. The discoveries of continuations. *LISP and Symbolic Computation* 6, 3/4 (Dec. 1993), 233–247. Special Issue on Continuations (Part I).

Amr Sabry and Matthias Felleisen. 1993. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation* 6, 3/4 (Dec. 1993), 289–360. Special Issue on Continuations (Part I).

Olin Shivers and David Fisher. 2006. Multi-return function call. *Journal of Functional Programming* 16, 4/5 (July/September 2006), 547–582. https://doi.org/10.1017/S0956796806006009

Guy L. Steele Jr. 1976. *Lambda, the ultimate declarative*. AI Memo 379. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Guy L. Steele Jr. 1978. *RABBIT: A Compiler for SCHEME*. Master's thesis. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts. Technical report AI-TR-474.