No-Brainer CPS Conversion

A functional pearl

Milo Davis William Meehan Olin Shivers

Northeastern University

{mdavis,wmeehan,shivers}@ccs.neu.edu

Abstract

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 fewer administrative redexes than any other linear-time algorithm of which we are aware. In fact, the output term is a normal form in a reduction system that generalises the notion of "adminstrative redexes" to what we call no-brainer redexes", that is, redexes whose reduction shrinks the size of the term. Of the four possible types of no-brainer redexes, three simply do not occur in the result. We state the theorems which establish the algorithm's desireable properties, along with sketches of the full proofs.

1. Introduction

Continuation Passing Style (CPS) is a variant of λ-calculus which has both theoretical and practical uses. CPS has two advantages over standard λ-calculus: it fixes evaluation order and names all intermediate computations. In fact, once the evaluation order is fixed by CPS, it is invariant to switches between call-by-name (CBN) and call-by-value (CBV) semantics. Plotkin used this property of CPS to prove a simulation between these two evaluation strategies [7].

When proving this simulation, Plotkin had to contend with additional complexity from "administrative redexes"; additional redexes introduced during the conversion of the direct-style (DS) term that are not present in the original term. To deal with this issue, Plotkin used the Colon transformation which effectively handles the issue in his theoretical context.

Less theoretical uses of CPS require terms to actually be reduced because even simple terms can grow significantly during the conversion process if administrative redexes are not eliminated. We provide an example DS term and the result of converting it with different algorithms below to illustrate the cost of administrative redexes and the effectiveness of previous conversion algorithms. Note that for clarity we use the standard λ-calculus in this section, but we switch to a scheme-like language in Section 2.

```
DS Term
(λ x.x)  (λ x.x)
Fisher/Reynolds algorithm
λ k1.(λk2.k2 (λx.λk3.k3 x))
λ m.(λk2.k2 (λx.λk3.k3 x)) λ n. m n k1
```

After Steele noted that CPS made a good intermediate language for compilers [10], a series of transformations were developed that yielded fewer and fewer administrative redexes. Algorithms like the Danvy/Filinski algorithm [5] are able to generate syntax free from administrative redexes, which allow them to generate smaller terms than the naive algorithm.

DEFINITION 2.1 (DS syntax).

$dsvar \in DSVAR ::= y$		*DS Variable reference*
$e \in DS ::= dsvar$		
	$\mid$ `(fun` y e`)`	*Function abstraction*
	$\mid$ `(`e e`)`	*Application*
	$\mid$ `(if` e e e`)`	*Conditional*

Figure 1. Direct-style Syntax

```
The Danvy/Filinski algorithm
λ k1. (λk2.k2 (λx.λk3.k3 x))
      (λk2.k2 (λx.λk3.k3 x)) k1
```

In this paper, we present a new CPS transformation for CBV λ-calculus, based on a defunctionalized version of the Danvy/Filinski algorithm. In linear time, we can produce a term free from administrative-redexes. Our algorithm also produces terms in the normal form of the No-Brainer reduction system, a superset of the administrative reductions that produces a strictly smaller term while preserving CBV semantics. Appel and Jim presented a similar set of reductions that was used in the SML/NJ compiler during the lexical analysis phase in [2]. However, their algorithm operated on terms after they were transformed to CPS whereas ours combines these two phases into a single transformation. The benefits of these additional reductions can be seen when converting our example term with our transformation.

```
Smart algorithm
 λ k1.k1 (λx.λk3.k3 x)
```

In Section 2 we present the scheme-like language we are using instead of the pure λ-calculus. In Section 3 we introduce the No-Brainer reduction system and prove its various properties. Section 4 contains a derivation of our Smart algorithm from the standard Fisher/Reynolds algorithm. This algorithm implements the No-Brainer reduction system. We then state theorems about its correctness and provide proof sketches. In this section we also introduce a new type of syntax constructor which we use to simplify our transformation. Section 5 includes a discussion of a series of variations and extensions that help scale our core calculus to a fully-featured programming language. In Section **??**, we conclude with remarks about our algorithm and directions for further work.

2. Notation

We have elected to use a scheme-like language as an alternative to pure λ-calculus. This gives us a clear separation between our

DEFINITION 2.2 (CPS syntax).

$triv \in TRIV$	::= x	*CPS Var reference*
	| `(lam (`x k`)` p`)`	*λ-term*
$p \in CPS$	::= `(call` $triv$ $triv$ $cont$`)`	*Function app*
	| `(ret` $cont$ $triv$`)`	*Cont app*
	| `(if` $triv$ p p`)`	*Conditional*
	| `(letc (`k $cont$`)` p`)`	*Cont binding*
$cont \in ABS\text{-}CONT$	::= k	*Cont var ref*
	| `(cont` x p`)`	*Cont λ-term*
	| `halt`	*Halt constant*

Figure 2. CPS Syntax

DS and CPS syntaxes and allows us to clearly follow the stack operations performed by the compiler. That being said, Both our Direct-Style (Figure 1) and CPS syntaxes (Figure 2) can be easily expressed in the standard λ-calculus. To make our language a more realistic model of a functional language, we have included `if` as a syntactic form. We also have included the constant `halt` which returns the value produced by the last expression in CPS. We also have an operation $nref$ which takes a variable and an expression and returns the variable's occurence count in the given term. For example, $nref(y, \texttt{(fun } y_2 \ y\texttt{)}) = 1$. Note that we use $nref$ to count occurences in both DS and CPS terms. Finally, we have the standard function FV which takes a term and returns the set containing its free variables.

The additional reason we have included `if` is to demonstrate the propper way to avoid duplicating continuations during its conversion. If the conversion of `if`, when improperly handled, leads to code explosion when the continuation is duplicated in both the then and else branches. This effect is worsened with nested `if` expressions. Below we present an example of this issue.

```
((if x y z) (g false))
```

Incorrectly converting this can cause the continuation to the conditional to be replicated down both arms of the result CPS conditional:

```
(if x
    (ret (cont f (g false (cont a (f a halt))))
         y)
    (ret (cont f (g false (cont a (f a halt))))
         z))
```

We must let-bind the continuation to rule out the possibility of exponential term growth:

```
(letc (k (cont f
           (call g false (cont a (f a halt)))))
  (if x (ret k y) (ret k z)))
```

Our language evaluates with semantics relatively similar to those of standard CPS. We have two application forms, `call` and `ret`. The first step when we reduce these forms is the same: we substitute the trivial term for the function's first parameter in the function's body. However, `lam` terms step to `letc` with the continuation bound to the function's second parameter. From there, a second k for $cont$ substitution finishes the evaluation of the `call`. The `if` form evaluates as expected, and `halt` returns the final value of the program, terminating the computation.

3. The No-Brainer Reduction System

Inlining is an important optimization performed by compilers. We use abstract reduction systems as a convenient formalization for inlining. Though highly optimizing compilers may have complicated heuristics for reduction, we address only a few relatively simple rewrite rules.

The intuition behind the No-Brainer reduction system is that we remove unnecessary bindings and inline values that will not increase the overall term size. For instance, replacing one variable with another will not change the term's size because all variables are the same size, and we can easily inline constants as long as their term size is as small as variables. We also want to inline function abstractions, especially because this can lead to reduction cascades, in which we can create more potential redexes where we inline the definition. However, if we do this in an undisciplined way, we may cause the size of our program to grow exponentially or cause our optimizations not to terminate. We can avoid both issues by inlining only when the variable to which a `lam` or `cont` term is bound occurs once. We include η-reduction as another way to decrease term size. We consider these reductions to be "No-Brainers" because they are all simple transformations that reduce term size and because they are obvious wins for the end user.

No-brainer reductions are semantics-preserving and can be applied to both CPS and DS terms. No-brainer redexes are designed to generalize the idea of "administrative reductions", so a term in No-Brainer normal form (NBNF) will also be in a normal form with respect to administrative reductions. For the purposes of our algorithm, we are only concerned with performing these reductions in CPS, so we present the reduction system and the proofs of its properties in our CPS Syntax. For a formal definition of No-Brainer Reduction, see Figure 3.

This reduction system is similar to the "shrinking-inlining" system created by Appel and Jim [2] for the SML/NJ compiler. However, we include η-reduction and omit their "dead-variable-elimination rule", which eliminates functions abstractions bound to variables when the variable does not appear in the term. We would not break normalization or confluence by introducing this rule, but we would lose the ability to reach a normal form with a constant number of passes over the syntax tree. Appel and Jim showed that a small number of passes eliminates the vast majority of these redexes, though we cannot identify and eliminate them all without an algorithm slower than linear time. The consequences of extending the No-Brainer system with this additional optimization rule are discussed further in Section 5.

When developing heuristics for inlining, it is important that the rules have two properties: strong normalization and confluence. A reduction system is strongly normalizing when there is no infinite series of reductions that can be applied to a term. The system is confluent if a given term can reach some reduced term no matter which reduction is taken first. While β-reduction, the traditional operational semantics of the λ-calculus, is confluent, it is not strongly normalizing. This means that reduction is not guaranteed to terminate, as can easily be demonstrated by the Ω combinator, but when it does terminate, the result will be the same regardless of the order of reductions performed. The restrictions we have placed on β-reduction will make it strongly normalizing. In Section 3.1, we will prove that the No-Brainer system has both of these properties.

3.1 Proofs of Properties

NBR's greediness characterizes it as a local optimization, which allows us to easily produce smaller terms than previous reductions on CPS terms. Normalization provides us with a guarantee that an implementation of the system will not cause an infinite reduction path, and confluence allows us to be unconcerned with the order in which we carry out reductions.

DEFINITION 3.1 (β_{cv}). *β-reduction where the argument is a consant or variable*

- $\texttt{(call (lam (}x_1\ k\texttt{) }p\texttt{) }x_2\ c\texttt{)} \underset{\beta cv}{\longrightarrow} \texttt{(letc (}k\ c\texttt{) }p[x_1 \mapsto x_2]\texttt{)}$
- $\texttt{(call (lam (}x\ k_1\texttt{) }p\texttt{) }t\ k_2\texttt{)} \underset{\beta cv}{\longrightarrow} \texttt{(ret (cont }x\ p[k_1 \mapsto k_2]\texttt{) }t\texttt{)}$
- $\texttt{(ret (cont }x_1\ p\texttt{) }x_2\texttt{)} \underset{\beta cv}{\longrightarrow} p[x_1 \mapsto x_2]$
- $\texttt{(letc (}k_1\ k_2\texttt{) }p\texttt{)} \underset{\beta cv}{\longrightarrow} p[k_1 \mapsto k_2]$

DEFINITION 3.2 ($\beta_{\lambda 1}$). *β-reduction where the argument is a lambda term and the bound variable occurs exactly once in its scope*

- $\texttt{(call (lam (}x_1\ k_1\texttt{) }p_1\texttt{) (lam (}x_2\ k_2\texttt{) }p_2\texttt{) }c\texttt{)} \underset{\beta\lambda 1}{\longrightarrow} \texttt{(letc (}k_1\ c\texttt{) }p_1[x_1 \mapsto \texttt{(lam (}x_2\ k_2\texttt{) }p_2\texttt{)}]\texttt{)}$ *where* $nref(x_1, p_1) = 1$
- $\texttt{(call (lam (}x_1\ k_1\texttt{) }p_1\texttt{) }t\texttt{ (cont }x_2\ p_2\texttt{))} \underset{\beta\lambda 1}{\longrightarrow} \texttt{(ret (cont }x_1\ p_1[k_1 \mapsto \texttt{(cont }x_2\ p_2\texttt{)}]\texttt{) }t\texttt{)}$ *where* $nref(k_1, p_1) = 1$
- $\texttt{(ret (cont }x\ p_1\texttt{) (lam (}x_2\ k\texttt{) }p_2\texttt{))} \underset{\beta\lambda 1}{\longrightarrow} p_1[x \mapsto \texttt{(lam (}x_2\ k\texttt{) }p_2\texttt{)}]$ *where* $nref(x, p_1) = 1$
- $\texttt{(letc (}k\texttt{ (cont }x\ p_2\texttt{)) }p_1\texttt{)} \underset{\beta\lambda 1}{\longrightarrow} p_1[k \mapsto \texttt{(cont }x\ p_2\texttt{)}]$ *where* $nref(k, p_1) = 1$

DEFINITION 3.3 (η). *classic η-reduction*

- $\texttt{(lam (}x\ k\texttt{) (call }t\ x\ k\texttt{))} \underset{\eta}{\rightarrow} t$ $\quad x, k \notin FV(t)$
- $\texttt{(cont }x\texttt{ (ret }c\ x\texttt{))} \underset{\eta}{\rightarrow} c$ $\quad x \notin FV(c)$

Figure 3. The No-Brainer Reduction System

DEFINITION 3.4 (Size of CPS Terms).

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

Figure 4. Definition of term size in our CPS notation

The following proofs are modeled after those in Barendregt [3]. Similar proofs can also be found in [2].

THEOREM 3.1 (Term Size Optimization).

$$\forall p_1, p_2 \in CPS, \qquad p_1 \rightarrow p_2 \implies size(p_1) > size(p_2)$$

Proof Sketch: By case analysis on the definition of no-brainer reduction, where size is defined as in Figure 4.

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 operational semantics.

THEOREM 3.2 (Strong Normalization).

$$\forall p \in CPS, \quad \nexists \text{ an infinite reduction sequence } p \rightarrow p' \rightarrow p'' \rightarrow \ldots$$

Proof Sketch: By Theorem 3.1 and structural induction on the term, using term size as a measure.

We now have a guarantee that there are no infinite reduction paths in the no-brainer system. Thus, we can implement the system algorithmically without having to worry whether the optimizations will fail to terminate. This also means that we can ensure that our algorithm's output is an NBNF. If this were not the case, our output might have been partially reduced, but we would not know if we had correctly eliminated all No-Brainer redexes.

While classic $\beta\eta$-reduction is known to be confluent, we must prove this property for the no-brainer system. We will accomplish this by proving each individual reduction of the system is confluent, and then demonstrating that they commute.

LEMMA 3.3 (Local Confluence of β_{cv} and $\beta_{\lambda 1}$).

$$\forall p, p', p'' \in CPS \text{ s.t. } p \underset{\beta cv}{\longrightarrow} p' \wedge p \underset{\beta cv}{\longrightarrow} p'', \quad \exists p''' \in CPS \text{ with } p' \underset{\beta cv}{\longrightarrow}^* p''' \wedge p'' \underset{\beta cv}{\longrightarrow}^* p'''$$

$$\forall p, p', p'' \in CPS \text{ s.t. } p \underset{\beta\lambda 1}{\longrightarrow} p' \wedge p \underset{\beta\lambda 1}{\longrightarrow} p'', \quad \exists p''' \in CPS \text{ with } p' \underset{\beta\lambda 1}{\longrightarrow}^* p''' \wedge p'' \underset{\beta\lambda 1}{\longrightarrow}^* p'''$$

Proof Sketch: Styled after a proof from Barendregt [3]. For each reduction, we inductively define a relation with the intention that the reduction is its transitive closure. So we prove β_{cv} and $\beta_{\lambda 1}$ are locally confluent by proving that the relations are both locally confluent.

LEMMA 3.4 (Commutativity of β_{cv} and $\beta_{\lambda 1}$).

$$\forall p \in CPS, \forall p', p'' \text{ s.t. } p \underset{\beta cv}{\longrightarrow} p', p \underset{\beta\lambda 1}{\longrightarrow} p'' \quad \exists\, p''' \text{ s.t. } p' \underset{\beta\lambda 1}{\longrightarrow}^* p''' \wedge p'' \underset{\beta cv}{\longrightarrow}^* p'''$$

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. (See Figure 6.)

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

LEMMA 3.5 (Confluence of No-Brainer β-reduction).

$$\forall p, p', p'' \in CPS \text{ s.t. } p \underset{\beta}{\rightarrow} p' \wedge p \underset{\beta}{\rightarrow} p'', \quad \exists p''' \in CPS \text{ with } p' \underset{\beta}{\rightarrow}^* p''' \wedge p'' \underset{\beta}{\rightarrow}^* p'''$$

Proof Sketch: By Lemma 3.3, Lemma 3.4, and Proposition 3.3.5 in Barendregt [3].

Classical η-reduction is known to be confluent ([3], Theorem 3.3.7), 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.

DEFINITION 3.5 (Direct-Style Reduction Context).

$$
\begin{aligned}
C_{DS} ::= {} & [\cdot] \\
& \mid \texttt{(fun } y\ C_{DS}\texttt{)} \\
& \mid \texttt{(}C_{DS}\ e\texttt{)} \mid \texttt{(}e\ C_{DS}\texttt{)} \\
& \mid \texttt{(if } C_{DS}\ e\ e\texttt{)} \mid \texttt{(if } e\ C_{DS}\ e\texttt{)} \\
& \mid \texttt{(if } e\ e\ C_{DS}\texttt{)}
\end{aligned}
$$

DEFINITION 3.6 (CPS Reduction Context).

$$
\begin{aligned}
C_{CPS} ::= {} & [\cdot] \\
& \mid \texttt{(lam (}x\ k\texttt{)}\ C_{CPS}\texttt{)} \\
& \mid \texttt{(call } C_{CPS}\ t\ c\texttt{)} \mid \texttt{(call } t\ C_{CPS}\ c\texttt{)} \\
& \mid \texttt{(call } t\ t\ C_{CPS}\texttt{)} \\
& \mid \texttt{(ret } C_{CPS}\ t\texttt{)} \mid \texttt{(ret } c\ C_{CPS}\texttt{)} \\
& \mid \texttt{(if } C_{CPS}\ p_1\ p_2\texttt{)} \\
& \mid \texttt{(if } t\ C_{CPS}\ p\texttt{)} \mid \texttt{(if } t\ p\ C_{CPS}\texttt{)} \\
& \mid \texttt{(letc (}k\ C_{CPS}\texttt{)}\ p\texttt{)} \mid \texttt{(letc (}k\ c\texttt{)}\ C_{CPS}\texttt{)}
\end{aligned}
$$

DEFINITION 3.7 (Reduction Inference Rules).

$$\frac{p_1 \to p_2}{C[p_1] \to C[p_2]} \qquad \frac{t_1 \to t_2}{C[t_1] \to C[t_2]} \qquad \frac{c_1 \to c_2}{C[c_1] \to C[c_2]}$$

Figure 5. Reduction in Context

LEMMA 3.6 (Commutativity of β and η).

$$\forall p \in \mathit{CPS}, \forall p', p''\ \mathit{s.t.}\ p \underset{\beta}{\to} p', p \underset{\eta}{\to} p'' \quad \exists\, p'''\ \mathit{s.t.}\ p' \underset{\eta}{\to}{}^* p''' \wedge p'' \underset{\beta}{\to}{}^* p'''$$

Proof Sketch: Diagrammatically using reduction context, as in our proof of Lemma 3.4. We consider every possible η-redex, performing β-reduction in context and subexpressions.

THEOREM 3.7 (Global Confluence of NBR).

$$\forall p \in \mathit{CPS}, \forall p', p'' \in \mathit{CPS}\ \mathit{with}\ p \to p', p \to p'', \quad \exists p''' \in \mathit{CPS}\ \mathit{with}\ p' \to^* p''' \wedge p'' \to p'''$$

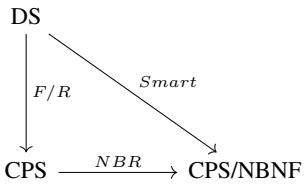
Proof Sketch: By Lemma 3.5, confluence of η-reduction, and Lemma 3.6.

Now that we have proven the no-brainer system normalizing 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 have to worry whether it will lead to a different translated term.

These reductions form a strong foundation for a compiler's inlining mechanism. As these reductions also do not require complex analyses to identify, we can implement these reductions during the transformation to CPS. In Section 4, we show how to do exactly that.

4. The Algorithm

We will systematically derive out transformation from a version of the Fisher/Reynolds algorithm translated into our syntax (Figure 7). The Fisher/Reynolds algorithm is garenteed to produce a valid CPS term, but performs no reductions. It is possible to reduce the algorithm's output after it has generated syntax, but that would require us to generate the larger term before another pass over the source tree created our desired result. This approach is costly in terms of both time and memory, so instead we seek to fuse those two passes into a single, linear-time transformation. The relationship between these approaches is shown in the commutativity diagram below.



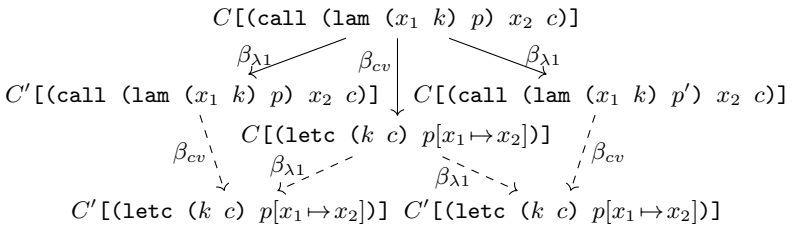
The derivation of our transformation takes place in two stages. In the first stage, we will define a Dumb algorithm with the same structure as our Smart algorithm, but that performs no reductions and produces terms α-equivalent to the Fisher/Reynolds algorithm. In the second stage, we will change the operations of the Dumb algorithm to those of the Smart algorithm.

We convert the Fisher/Reynolds algorithm into the Dumb algorithm by abstracting over the syntax generation mechanisms of the Fisher/Reynolds algorithm. After we change the data representation of syntax during the conversion, it becomes significantly easier for us to perform reductions.

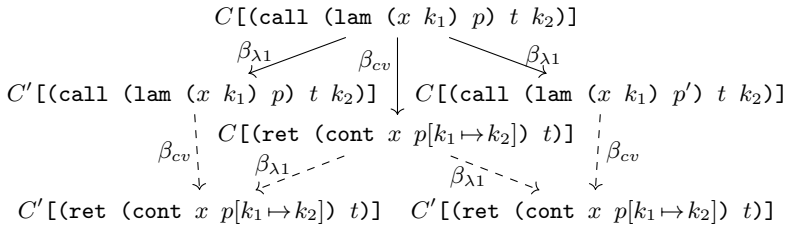
First, we abstract over the generation of continuation syntax. The essence of the CPS transformation is the reordering of terms so that the results of nested applications from the DS term are bound to continuation variables. We use a data structure that captures the same information as syntactic continuations of the Fisher/Reynolds algorithm to save the outer applications until we are finished converting their dependencies. Our data structure takes one of five possible forms, all from cases of the Fisher/Reynolds algorithm. In the application case of Fisher/Reynolds algorithm, we can identify two types of continuations: those waiting for a function and those waiting for a variable. There is an additional continuation type in the `if` case: the continuation waiting for a boolean value. Finally, there are continuation variables introduced by `cont` and `letc` and the base `halt` continuation. With these observations, we can convert these pieces of syntax into a data structure that captures the same information. In our more complex algorithm, we will use case analysis on the terms that would be bound to the continuation's parameter to reveal reduction opportunities. The ability to perform this case analysis is the core reason for us to convert continuation syntax into a data structure.

As our data structure will contain DS terms in three of our five cases, we are forced to contend with the issue of variable freshness. When dealing solely in mathematics, asserting a variable's freshness in a side condition is sufficient, but a practical implementation of a CPS algorithm will almost always require the replacement of DS variables to ensure no capture occurs. This is usually done using a symbol table which provides a partial mapping from DS variables to their CPS counterparts. In the Smart algorithm, this symbol table is essential for reductions, but for now, its only purpose is the generation of fresh variables during the conversion.

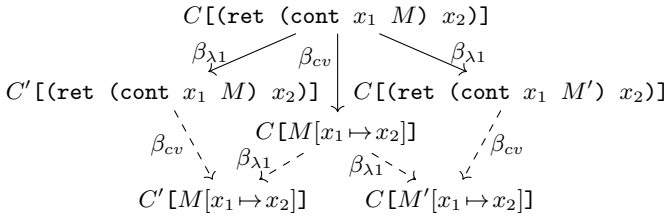
Abstracting this syntactic information into a data structure will help us when we perform reductions, but it does us little good unless we can convert the data structure back into syntax. To turn this data structure into syntax, we define a function $bless_{cd}$, used when we put a continuation in its *final resting place*. This function, notated $\widetilde{c}^{c}$, converts a continuation data structure into syntax. Both the data structure, and the function to convert it into syntax are defined in Figure 8.



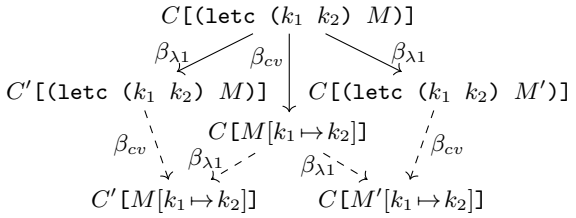
(a) Commutativity of $\beta_{\lambda 1}$ with the first β_{cv} rule



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



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



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

Figure 6. Commutativity of β_{cv} and $\beta_{\lambda 1}$

DEFINITION 4.1 (The Fisher/Reynolds algorithm).

$$P\ e\ cont \triangleq \begin{cases} \texttt{(ret } cont\ x\texttt{)} & e = y \\ \texttt{(ret } cont\ \texttt{(lam (}x\ k\texttt{) (}P\ e\ k\texttt{)))} & e = \texttt{(fun } y\ e\texttt{)},\ x \textit{ fresh} \\ \texttt{(}P\ e_f\ \texttt{(cont } x_f\ \texttt{(}P\ e_a\ \texttt{(cont } x_a\ \texttt{(call } x_f\ x_a\ cont\texttt{)))))} & e = \texttt{(}e_f\ e_a\texttt{)}, x_f \textit{ fresh} \\ \texttt{(letc (}x\ cont\texttt{) (}P\ e_1\ \texttt{(cont } x_b\ \texttt{(if } x_b\ \texttt{(}P\ e_2\ k\texttt{) (}P\ e_3\ k\texttt{)))))} & e = \texttt{(if } e_1\ e_2\ e_3\texttt{)},\ k \textit{ fresh} \end{cases}$$

Figure 7. The Fisher/Reynolds algorithm

$$\begin{array}{lll} c \in \text{ABS-CONT} ::= \texttt{halt} & & \texttt{halt} \\ \quad \mid k & & k \\ \quad \mid FCont(e, E, c) & \texttt{(cont } x_a\ \texttt{(ret (cont } x_a\ \texttt{(}P_E\ e\ E\ ACont(x_a, c)\texttt{)) } x_f\texttt{))} & \\ \quad \mid ACont(a, c) & \texttt{(cont } x_f\ \texttt{(call } x_f\ \widetilde{a}^v\ \widetilde{c}^c\texttt{))} & \\ \quad \mid ICont(e_1, e_2, E, c) & \texttt{(cont } x_b\ \texttt{(letc (}j\ \widetilde{c}^c\texttt{) (if } x_b\ \texttt{(}P_E\ e_1\ E\ j\texttt{) (}P_E\ e_2\ E\ j\texttt{))))} & \end{array}$$

Figure 8. Continuation Data Structure

$$\begin{array}{lll} a \in \text{ABS-TRIV} ::= x & & x \\ \quad \mid \langle \texttt{(fun } y\ e\texttt{)}, E \rangle & \texttt{(lam (}x\ k\texttt{) (}P_E\ e\ E[y \mapsto x]\ k\texttt{))}\ x,\ k \text{ fresh} & \end{array}$$

Figure 9. Abstracted Trivial

DEFINITION 4.2 (The extended Fisher/Reynolds algorithm).

$$P_E\ e\ E\ c \triangleq \begin{cases} \texttt{(ret } \widetilde{c}^{c}\ \widetilde{E[y]}^{v}\texttt{)} & e = y \\ \texttt{(ret } \widetilde{c}^{c}\ \texttt{(lam (}x\ k\texttt{) (}P_E\ e\ E[y \mapsto x]\ k\texttt{)))} & e = \texttt{(fun } y\ e\texttt{)},\ x,\ k \textit{ fresh} \\ (P_E\ e_f\ E\ FCont(e_a, E, c)) & e = \texttt{(}e_f\ e_a\texttt{)}, x_f \textit{ fresh} \\ \texttt{(letc (}j\ \widetilde{c}^{c}\texttt{) (}P_E\ e_1\ ICont(e_2, e_3, E, j)\ \texttt{))} & e = \texttt{(if } e_1\ e_2\ e_3\texttt{)},\ j \textit{ fresh} \end{cases}$$

Figure 10. The Fisher/Reynolds algorithm with environments and continuation data structures

When we call P on the initial DS term, our algorithm proceeds by two mutually recursive steps. First, we shift the direct-style term into the continuation until we reach a trivial DS value. Next, we reify the continuation, which may require us to return to the first phase of the algorithm. We repeat this process until the algorithm terminates, which it is guaranteed to do because each recursive call reduces either the DS term or the continuation data structure.

When we perform reductions, `fun` terms might be reified in a different context than where they begin in the source term. To avoid variable capture, we package up these "delayed" user functions with an additional symbol table. Now each user function carries its context with it as we move it through the algorithm. This data structure, (notated $\langle\texttt{(fun } y\ e\texttt{)}, E\rangle$) which we call a "static closure", corresponds to the data structure which we use to delay continuations.

We manipulate static closures in much the same way we manipulate the continuation data structure. When we first encounter a function abstraction, we pair it with its environment, introducing a static closure. Next, we move it around the algorithm until it reaches its final location. In the Dumb algorithm, this movement is trivial, however in the Smart algorithm, we might move a static closure multiple times before we bless it using the function $bless_v$. As the No-Brainer β-reduction rules also address variables, we create a variant type comprised of static closures and CPS variables. We extend $bless_v$ to produce syntax from CPS variables by returning them unchanged. We say that an abstracted trivial or continuation has been blessed when it has been converted to concrete syntax by $bless_v$ or $bless_c$. The bless function for values, notated $\widetilde{a}^{v}$, is defined next to each variant of the abstract-trivial data type in Figure 9.

To inline abstract trivial terms, we change the domain of our symbol table from CPS variables. The type of a symbol table is now (E : DSVAR $\rightharpoonup$ ABS-TRIV). We use the term "static environment" to refer to these symbol tables that map DS variables to arguments. There are two operations we can perform on environments: looking up a variable in the environment and extending the environment with a new binding. These operations are notated $E[y]$ and $E[y \mapsto a]$.

To avoid any variable name collisions, when a DS variable is introduced, we create a new entry in our environment which maps that DS variable to a fresh CPS version. We consider a variable to be fresh if it does not occur in the current term and does not occur in the environment. When we encounter a variable reference in a context other than its introduction, we perform a lookup operation on the variable and replace it with the value in the environment. This means that any value we place in the environment will replace the DS variable to which it is bound. As the values returned from *lookup* might be static closures, when they reach their final placement, we use $bless_v$ to convert them into concrete syntax. In the Smart algorithm, we often feed the results of *lookup* into other functions, creating cascades of reductions. These operations correspond to the environment model of λ-calculus β-reduction. This is in contrast to the substitution model of β-reduction used in Section 3.

$$\begin{aligned} C &: \text{DS} \rightarrow \text{ENV} \rightarrow \text{ABS-CONT} \rightarrow \text{CPS} \\ Ret &: \text{ABS-CONT} \rightarrow \text{ABS-TRIV} \rightarrow \text{CPS} \\ Call &: \text{ABS-TRIV} \rightarrow \text{ABS-TRIV} \rightarrow \text{ABS-CONT} \rightarrow \text{CPS} \\ bless_c &: \text{ABS-CONT} \rightarrow \text{ABS-CONT} \\ bless_v &: \text{ABS-TRIV} \rightarrow \text{TRIV} \\ If &: \text{DS} \rightarrow \text{DS} \rightarrow \text{ENV} \rightarrow \text{ABS-CONT} \rightarrow \text{CPS} \end{aligned}$$

Figure 11. Constructor Signatures

We compose these new functions to form a version of the Fisher/Reynolds algorithm which utilizes static environments to replace user variables with their CPS equivalents. We do not yet place static closures in the environment as this would β-reduction them. To differentiate the versions of the algorithm, we rename the toplevel function of this new algorithm P_E. This new version of the algorithm is defined in Figure 10.

Though this algorithm will produce terms in the same style as the original Fisher/Reynolds algorithm, the introduction of environments weakens our guarantee that a valid CPS term will be produced. Because *lookup* is only partial, our algorithm will error when it encounters a variable it has not seen. Without an additional precondition, any theorem about the algorithms output is predicated on its termination without a failed lookup. We define a well-formedness property for term-environment pairs and an analogous property for continuations so that when we convert a term with this property, *lookup* will not error. Theorem 4.1 proves that this property is preserved on all recursions of all algorithms we define that use the environment.

DEFINITION 4.3 (Environment Well-formedness property).

$$(e, E) \in W \textit{ iff } FV(e) \subset domain(E) \wedge \forall r \in range(E),\ r \in W \vee r \in VAR$$

DEFINITION 4.4 (Well-formed continuation).

$$c \in WC \textit{ iff } \begin{cases} k & \\ FCont(e, E, c) & (e, E) \in W \wedge c \in WC \\ ACont(a, c) & (a \in W \vee a \in VAR) \wedge c \in WC \\ ICont(e_1, e_2, E, c) & (e_1, E),\ (e_2, E) \in W \wedge c \in WC \end{cases}$$

4.1 The Dumb algorithm

At this point, we have created almost all of the machinery we will need to perform reductions and it is time to add the final touches that give us the Dumb algorithm. But as we split off different pieces of the transformation into different data structures, the P function became increasingly monolithic. To clean up our implementation, we introduce a series of syntax constructors which transform delayed values and DS syntax into CPS, optionally reducing the term

in the process. For clarity, we name each constructor for the syntax it produces.

For now, this means that the syntactic term produced will share the name of the constructor that produced it e.g. the Ret constructor produces a `ret` form, the $Call$ constructor produces a `call` form, and so on. The type signatures for the constructors and the other functions used by our algorithms can be found in Figure 11. In the full Smart algorithm, constructors will also perform reductions, so the syntax they generate will be the No-Brainer normal form of the syntax for which they are named. Instead of the Ret constructor generating a `ret` form, it will produce a term that is the No-Brainer reduction of a naively generated `ret` form. The naming of the smart constructors enhances the connections between the unreduced CPS term and Smart-algorithm output. our algorithm as we add more complicated operations. The following example of a β_{cv} redex demonstrates the difference between the Dumb algorithm and Smart algorithm. Recall that extending the environment is equivalent to an y for $\widetilde{a}^v$ substitution.

Dumb algorithm

$$(Call_d\ \langle(\texttt{fun}\ y\ e), E\rangle\ x\ k) = (\texttt{call}\ \widetilde{\langle(\texttt{fun}\ y\ e), E\rangle}^{\,v}\ x\ k)$$

Smart algorithm

$$(Call\ \langle(\texttt{fun}\ y\ e), E\rangle\ x\ k) = (C_s\ e\ E[y \mapsto x]\ k)$$

We call the constructors that perform reductions "smart-constructors" With the use of constructors instead of the single P function, there is only one artifact remaining from the original Fisher/Reynolds algorithm algorithm: the name of the toplevel function. We address this by renaming the toplevel function from P to C. The full Dumb algorithm is defined in Figure 12.

Each function in the Dumb algorithm has an analog in the Smart algorithm. To differentiate them, we use a subscript "d" and "s". We omit these subscripts when it is clear from context to which algorithm we are referring. We also change our notation for $bless_{vd}$ and $bless_{cd}$ in the Smart algorithm, notating the $bless_v$ and $bless_c$ functions $\overline{a}^{\{v|c\}}$. These functions generate concrete syntax from the representative data structure, so, after a value has been blessed, we cannot perform further reductions on it. Therefore, we only bless a value when we put it in its final resting place.

As our derivation shows, this algorithm is equivalent to the Fisher/Reynolds algorithm. Therefore, we know that executing its output with standard CPS semantics will simulate the original direct style term. However, this algorithm contains all of the administrative-redexes we promissed to address and is more complicated than when we began. We present it to make the essential concepts of our algorithm clear before we increase complexity in the Smart algorithm.

4.2 The Smart algorithm

The Smart algorithm operates in two passes. The first pass identifies the No-Brainer β-redexes in the source term and the second converts the term into CPS. Our algorithm uses a count of a function's parameter's occurrences in its body to efficiently tell if a function is β-reducible. Counting these occurrences is performed by a simple recursive pass over the source term. If a previous compiler pass has already made all variable names unique, the counts can be tracked in an external dictionary. If this step has not yet been performed, it is likely easier to annotate variable-introduction sites with occurrence counts. Either storage method can be chosen, so long as the $nref$ function operates as expected.

Identifying η-reduction opportunities is more complex and subtle. Our original, flawed method of identifying η-redexes examined the structure of the DS term and checked the DS variable reference counts. In the presence of our No-Brainer β-redexes, there are three issues: DS terms with no η-redex can reduce to η-reducible terms, variable counts can be duplicated by the $\beta_{\lambda 1}$ rule making the DS counts inaccurate, and a term that η-reduce to a variable may now be reducible by the β_{cv} rule.

Our first issue is captured by the following DS term that is ineligible for η-reduction: `(fun` y_1 `((fun` y_2 `(`y_2 y_1`))` f`))`. But, as the following reductions show, the other No-Brainer reductions quickly transform its CPS equivalent into a No-Brainer reduction opportunity.

$$\begin{aligned}
&\texttt{(lam (}x_1\ k_1\texttt{)}\\
&\quad\texttt{(call (lam (}x_2\ k_2\texttt{) (call }x_2\ x_1\ k_2\texttt{))}\ triv_f\ k_1\texttt{))}\\
&\rightarrow \texttt{(lam (}x_1\ k_1\texttt{) (call }triv_f\ x_1\ k_1\texttt{))}\\
&\underset{\eta}{\rightarrow} triv_f
\end{aligned}$$

Though we could perform reductions analogous to the No-Brainer CPS reductions on the source term, we can use our smart constructors that we already know perform reductions to produce the NBNF of the converted body Then, we perform a check on the converted form. Regardless of the result of this check, we will use all or part of the converted body in the blessed form. To avoid duplicating the conversion, we let-bind the result to avoid converting the function body twice.

Our second issue is that we may also change a variable's occurrence counts when applying β_{cv} reductions. Because we cannot rely on variable reference counts from the DS term, we use a new set of variable counts computed during the transformation of a term. This can be efficiently computed by adding a second statement to the body of $bless_v$. Whenever this function is called on a variable, the count of references to that variable in a global table is incremented. When a `lam` term of the form `(lam (`x`) (call` $triv_f$ x `))` is being considered for η-reduction, $nref(x, \texttt{(lam (}x\ k\texttt{) (call }triv_f\ x\ k\texttt{))}) = 1$. If the term has the structure that suggests η-reduction, x must occur in $triv_f$. Our usage of fresh variable names allows us to use a single global counts table instead of multiple local tables that require a linear time merge operation.

The final issue we encounter with η-reduction is caused by its interactions with β-reduction. In the following term, where x_f is a CPS variable and $nref(x_1, p) > 1$, there is no $\beta_{\lambda 1}$ redex until after η-reduction has been performed.

$$\begin{aligned}
&\texttt{(call (lam (}x_1\ k_1\texttt{) }p\texttt{)}\\
&\qquad\texttt{(lam (}x_2\ k_2\texttt{) (call }x_f\ x_2\ k_2\texttt{))}\\
&\qquad k_3\texttt{)}
\end{aligned}$$

To solve the issue, in the above case, the $Call$ constructor will bless the argument `lam` before it blesses the term in function call position. If the blessed argument is a variable, we extend the environment and proceed with β-reduction. If the argument is still a `lam` term, we reify the term in call position and generate the `call` form.

With an understanding of how to identify β-redexes and perform η-redexes, we can proceed to describe the intuition for performing our β-reductions. As we perform a substitution for everything in the environment, when we need to β-reduction, we extend the environment with a variable for argument mapping. This intuition is formalized in Lemma 4.3.

We do not use the environment to β-reduction continuations. This is because continuations do not occur in the source term, so we are creating completely new syntax. Therefore, the environment, which performs substitutions on the existing term, will not help us. However, inlining a continuation just requires us to continue passing the continuation deeper into the term until we unpack it and perform reductions on its components. As our algorithm already will use the environment to substitute variables and arguments into

$$C\ e\ E\ c \triangleq \begin{cases} (Ret_d\ c\ E[y]) & e = y \\ (Ret_d\ c\ \langle e, E\rangle) & e = \texttt{(fun}\ y\ e_2\texttt{)} \\ (C_d\ e_1\ E\ FCont(e_2, E, c)) & e = \texttt{(}e_1\ e_2\texttt{)} \\ (C_d\ e_1\ E\ ICont(e_2, e_3, E, c)) & e = \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \end{cases}$$

$$Ret\ c\ a \triangleq \texttt{(ret}\ \widetilde{c}^{c}\ \widetilde{a}^{v}\texttt{)}$$
$$Call\ a_1\ a_2\ c \triangleq \texttt{(call}\ \widetilde{a_1}^{v}\ \widetilde{a_2}^{v}\ \widetilde{c}^{c}\texttt{)}$$
$$If\ a\ e_1\ e_2\ E\ c \triangleq \texttt{(letc (}j\ \widetilde{c}^{c}\texttt{) (if}\ \widetilde{a}^{v}\ (C_d\ e_1\ E\ c)\ (C_d\ e_2\ E\ c)\texttt{))}$$

$$\widetilde{c}^{c} \triangleq \begin{cases} \texttt{halt} & c = \texttt{halt} \\ k & c = k \\ \texttt{(cont}\ x\ (C_d\ e\ E\ ACont(x, c_2))\texttt{)} & c = FCont(e, E, c_2) \\ \texttt{(cont}\ x\ (Call\ \widetilde{a}^{v}\ x\ \widetilde{c_2}^{c})\texttt{)} & c = ACont(a, c_2) \\ \texttt{(cont}\ x\ (If\ x\ e_1\ e_2\ E\ c_2)\texttt{)} & c = ICont(e_1, e_2, E, c_2) \end{cases}$$

$$\widetilde{a}^{v} \triangleq \begin{cases} x & a = x \\ \texttt{(lam (}x'\ k\texttt{) (}C_d\ e\ E[x \mapsto x']\ k\texttt{))} & a = \langle \texttt{(fun}\ y\ e\texttt{)}, E\rangle \end{cases}$$

Figure 12. The Dumb algorithm

continuation data structures while we reify them, so we will have no problem performing these reductions.

The Smart algorithm, similarly to the dumb version, begins with a call to the C function. This function uses the Ret constructor on values and adds to the continuation data structure while reducing the source term otherwise. In contrast to the Dumb algorithm, the Ret_s constructor does one of several things depending on the type of continuation it is passed. When the continuation is a variable, no reductions can be preformed and so a `ret` form is generated. If the continuation is an $FCont$, it converts the argument using C, shifting the argument form into the continuation data structure creating an $ACont$. When Ret is called on an ACont, the function is stored in the continuation data structure and the argument is provided to the Ret_s constructor.

At this point, the $Call$ constructor is capable of generating trivial values for all three parts of the function call using $bless_{vs}$ and $bless_{cs}$, but there may be an opportunity for β-reduction. If the first argument of the $Call$ is a static closure and the user function's parameter occurs once in its body, or if the argument it is passed is a constant or a variable, i.e. if the `call` is an instance of a No-Brainer β-reduction rule, we extend the environment of the static closure with a binding for the user variable and convert the body using the C_s constructor with the continuation argument from the $Call$. This can lead to cascades of inlining as terms are repeatedly taken out of the environment and then inlined again. If the argument to the call is a static closure and the parameter to the static-closure in call position occurs multiple times in its body, there is still a chance that reductions can be made if the argument η-reduce to a variable. In this case, we bless the argument first and let-bind it, extending the environment if it is a variable and generating a `call` form otherwise. If a reduction cannot be made, the unreducible term is let bound and the environment is extended with the variable to which it is bound. This completely removes `lam` expressions from function call position. They are either let-bound or their parameter is inlined, leaving only their body behind. The $bless_{cs}$ constructor works as expected in the variable case, and uses the Ret form to perform reductions otherwise. Similarly, the $bless_{vs}$ constructor uses the $Call$ form to reduce user functions.

4.3 Proofs of properties

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.

Before we can prove that our algorithm's output has these properties, we must prove that it terminates without producing an error. As we have only one function, *lookup*, which may error, we need to certify that all of uses of this function will succeed. This intuition is given for environments in Definition 4.3 and for continuations in Definition 4.4.

THEOREM 4.1 (Preservation of well-formedness).

$$\forall (e, E) \in W,\ c \in WC$$
$$(P_E\ e\ E\ c)\ \textit{preserves}\ (e, E) \in W\ \textit{and}\ c \in WC\ \textit{on all recursions}$$
and
$$(C_d\ e\ E\ c)\ \textit{preserves}\ (e, E) \in W\ \textit{and}\ c \in WC\ \textit{on all recursions}$$
and
$$(C_s\ e\ E\ c)\ \textit{preserves}\ (e, E) \in W\ \textit{and}\ c \in WC\ \textit{on all recursions}$$

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

The next two lemmas lead us to our theorem that the Smart algorithm is a reduction of the Dumb algorithm. We do this using two lemmas. Our first goal is to formalize the notion that when we recur with an extended environment, we are β-reducing the term. To prove this equivalence, we use substitution on unreified continuations. We define this property in Figure 14 and prove its equivalence to substitution on reified continuations in our first lemma.

$$C\ e\ E\ c \triangleq \begin{cases} (Ret\ c\ E[y]) & e = y \\ (Ret\ c\ \langle e, E\rangle) & e = \texttt{(fun}\ y\ e_2\texttt{)} \\ (C_s\ e_1\ E\ FCont(e_2, E, c)) & e = \texttt{(}e_1\ e_2\texttt{)} \\ (C_s\ e_1\ E\ ICont(e_2, e_3, E, c)) & e = \texttt{(if}\ e_1\ e_2\ e_3\texttt{)} \end{cases}$$

$$Ret\ c\ a \triangleq \begin{cases} \texttt{(ret}\ \overline{c}^c\ \overline{a}^v\texttt{)} & c = k \vee c = \texttt{halt} \\ (C_s\ e\ E\ ACont(a, c)) & c = FCont(e, E, c) \\ (Call\ a\ a_2\ c_2) & c = ACont(a_2, c_2) \\ (Ret\ ICont(e_1, e_2, c_2, E)\ a) & c = (If\ a_2\ e_1\ a_2\ E\ c_2) \end{cases}$$

$$Call\ a_1\ a_2\ c \triangleq \begin{cases} \texttt{(call}\ x\ \overline{a_2}^v\ \overline{c}^c\texttt{)} & a_1 = x \\ (C_s\ e\ E[y \mapsto a_2]\ c) & a_1 = \langle\texttt{(fun}\ y\ e\texttt{)}, E\rangle \wedge nref(y, e) = 1 \\ \text{let } blessed = \overline{a_1}^v \text{ in} & a_1 = \langle\texttt{(fun}\ y_1\ e_1\texttt{)}, E_1\rangle \wedge a_2 = \langle\texttt{(fun}\ y_2\ e_2\texttt{)}, E_2\rangle \\ \text{if } blessed = x & \\ \text{then } (C_s\ e\ E_1[y_1 \mapsto x]\ c) \quad \textit{(* New } \beta_{cv} \textit{ redex *)} & \\ \text{else } \texttt{(call}\ \overline{a_1}^v\ blessed\ \overline{c}^c\texttt{)} & \\ \texttt{(ret (cont}\ x'\ (C_s\ e\ E[y \mapsto x']\ c))\ \overline{a_2}^v\texttt{)} & a_1 = \langle\texttt{(fun}\ y\ e\texttt{)}, E\rangle \end{cases}$$

$$If\ a\ e_1\ e_2\ E\ c \triangleq \begin{cases} \texttt{(if}\ \overline{a}^v\ (C_s\ e_1\ E\ k)\ (C_s\ e_2\ E\ k)\texttt{)} & c = k \\ \texttt{(letc (}j\ \overline{c}^c\texttt{)}\ (If\ e_1\ e_2\ E\ j\ \texttt{))} & otherwise \end{cases}$$

$$\overline{c}^c \triangleq \begin{cases} \texttt{halt} & c = \texttt{halt} \\ k & c = k \\ \texttt{(cont}\ x\ (Ret\ c\ x)\texttt{)} & otherwise \end{cases}$$

$$\overline{a}^v \triangleq \begin{cases} x & a = x \\ \text{let } body = (C_s\ e\ E[y \mapsto x]\ k) \text{ in} & a = \langle\texttt{(fun}\ y\ e\texttt{)}, E\rangle,\ x, k \text{ fresh} \\ \text{if } nref(x, body) = 1 \text{ and } body = \texttt{(call}\ triv\ x\ k\texttt{)} & \\ \text{then } triv \quad \textit{(* } \eta \textit{ reduction *)} & \\ \text{else } \texttt{(lam (}x\ k\texttt{)}\ body\texttt{)} & \end{cases}$$

Figure 13. The Smart algorithm

DEFINITION 4.5 (Continuation Substitution).

$$\texttt{halt}[k \mapsto c] \triangleq \texttt{halt}$$

$$k'[k \mapsto c] \triangleq \begin{cases} c & k = k' \\ k' & k \neq k' \end{cases}$$

$$ACont(a, c')[k \mapsto c] \triangleq ACont(a, c'[k \mapsto c])$$

$$FCont(e, E, c')[k \mapsto c] \triangleq FCont(e, E, c'[k \mapsto c])$$

$$ICont(e_1, e_2, E, c')[k \mapsto c] \triangleq ICont(e_1, e_2, E, c'[k \mapsto c'])$$

Figure 14. The substitution $[k \mapsto c]$ is extended to operate on the elements of the *Continuation* domain, respecting their interpretation as concrete CPS terms.

LEMMA 4.2 (Continuation substitution is term substitution).

$$\forall c \in \mathit{WC},\ k,\ c' \in \mathit{WC},\ (e, E)\ \in \mathit{W},$$
$$\widetilde{c}^c[k \mapsto \widetilde{c'}^c] = \widetilde{c[k \mapsto c']}^c \wedge$$
$$(C_d\ e\ E\ c)[k \mapsto \widetilde{c'}^c] = (C_d\ e\ E\ c[k \mapsto c'])$$

Proof Sketch: By mutual induction across $bless_v$ and C and induction on the structure of the continuation and the structure of the e term. The proof proceeds by induction through the k and $ACont(a, c)$ cases, but the $FCont(e, E, c)$ 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 this continuation substitution property, we can prove that β-reduction is equivalent to environment extension.

LEMMA 4.3 (Environment extension is β-reduction).

$$\forall (e, E[y \mapsto a]) \in \mathit{W},\ c \in \mathit{WC},$$
$$(C_d\ e\ E[y \mapsto x]\ c)[x \mapsto \widetilde{a}^v]$$
$$= (C_d\ e\ E[y \mapsto a]\ c)$$
where x and k are fresh

Proof Sketch: By induction on the structure of the DS term with additional cases for variables not equal to y as function parameters and representing the entirety of e. We use the continuation substitution property verified in Lemma 4.2 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 $bless_c$ backwards. This allows us to propagate environment extensions into continuation data structures.

With these lemmas in hand, we can prove our initial theorem: that the Smart algorithm is a No-Brainer reduction of the Dumb algorithm.

THEOREM 4.4 (Dumb algorithm $\rightarrow^*$ Smart algorithm). *The output of the Dumb algorithm algorithm reduces to the output of the Smart algorithm:*

$$\begin{gathered}\forall (e, E) \in W, a \in W, c \in WC \\ (C_d\ e\ E\ c) \rightarrow^* (C_s\ e\ E\ c) \textit{ and} \\ (Ret_d\ c\ a) \rightarrow^* (Ret\ c\ a) \textit{ and} \\ (Call_d\ a\ a\ c) \rightarrow^* (Call\ a\ a\ c) \textit{ and} \\ (If_d\ a\ e\ e\ E\ c) \rightarrow^* (If\ a\ e\ e\ E\ c) \textit{ and} \\ \widetilde{a}^v \rightarrow^* \overline{a}^v \textit{ and} \\ \widetilde{c}^c \rightarrow^* \overline{c}^c\end{gathered}$$

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 4.3 is used to relate the two algorithms through β-reduction.

As a consequence, we have a proof that the Smart algorithm nbr-reduces to the Dumb algorithm. We also know that the Smart algorithm is a reduction of the Fisher/Reynolds algorithm CPS transform. This second fact tells us that there is a simulation between DS evaluation and the evaluation of our algorithm's output.

COROLLARY 4.5.

$$\forall e, FV(e) = \varnothing \implies (C_d\ e\ [\cdot]\ \texttt{halt}) \rightarrow^* (C_s\ e\ [\cdot]\ \texttt{halt})$$

Though the output of our algorithm may be valid CPS, the question of whether it is in No-Brainer normal form remains. A proof of this property is provided in Theorem 4.6.

THEOREM 4.6 (The Smart algorithm output is an NBNF).

$$\forall (e, E) \in W, a \in W, c \in WC$$

the following terms are in No-Brainer normal form

- $(C_s\ e\ E\ c)$
- $(Ret\ c\ a)$
- $(Call\ a_1\ a_2\ c)$
- $(If\ a\ e_1\ e_2\ E\ c)$
- $\overline{a}^v$
- $\overline{c}^c$

Proof Sketch: As Lemma 4.3 formalizes the notion that environment extension is equivalent to β-reduction, proving that the 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 $bless_v$. Finally, we must ensure that we have reduced continuations whenever possible. This requires us to examine the cases of Ret_s which shows that we do indeed delegate to smart constructors that perform these reductions where appropriate.

With slight modifications, Theorem 4.3 and Theorem 4.6 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.

In this section we have presented a new algorithm for CPS conversion. This algorithm in just two passes over the source tree, generates terms in No-Brainer normal form. We have unified the CPS conversion and inlining portions of the compiler, leaving a strictly smaller term which can undergo more complex static analyses. In our next section, we present a series of simple variations which may be used to extend this algorithm to do other optimizations during the translation into CPS.

5. Variations

We've focussed the development of our algorithm, so far, on the 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.

5.1 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, we can do constant propagation with these. 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 λ-expression.

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; the point is to do, judiciously, the easy things so that a normalising converter can clear away the "underbrush"[1] in a simple, linear-time way before proceeding to the more complex, costly transformation phases of a compiler.

5.2 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 policies [1]. 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 [9, 8].

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

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 (x y z) ...)` into

```
(fun (x) (fun (y) (fun (z) ...))).
```

That is, if the first term occurs in a β redex, we can "reach into" the middle of the parameter list and reduce away the `y` parameter, even if the first `x` parameter cannot be reduced. 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.

5.3 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 [6].

5.4 Metacontinuations

If we Church-encode the elements of the $Continuation$ set (that is, values constructed by $FCont$, $ACont$, *etc.*), then we can get an algorithm that uses the clever "metacontinuation" described by representation introduced by Danvy and Filinski [5]. 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 the space of functions.

Right cite?

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 [4] for purposes of verification (a task we are currently undertaking).

References

[1] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 2006.

[2] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(05):515–540, 1997.

[3] H. P. Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[4] R. S. Boyer and J. S. Moore. A theorem prover for a computational logic, 1990.

[5] O. Danvy and A. Filinski. Representing control: a study of the cps transformation, 1992.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[7] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[8] O. Shivers and D. Fisher. Multi-return function call. *ACM SIGPLAN Notices*, 39(9):79–89, 2004.

[9] O. Shivers and D. Fisher. Multi-return function call. *Journal of Functional Programming*, 16(4-5):547–582, 2006.

[10] G. L. Steele. Lambda: The ultimate declarative. Technical report, MIT AI Lab, Cambridge, MA, USA, 1976.