Call-by-Value Combinatory Logic and the Lambda-Value Calculus

John Gateley and Bruce F. Duba*

Rice University

Abstract

Since it is unsound to reason about call-by-value languages using call-by name equational theories, we present two by-value combinatory logics and translations from the λ-value (λ_v) calculus to the logics. The first by-value logic is constructed in a manner similar to the $\lambda_{\mathbf{v}}$-calculus: it is based on the by-name combinatory logic, but the combinators are strict. The translation is non-standard to account for the strictness of the input program. The second by-value logic introduces laziness to K terms so that the translation can preserve the structure of functions that do not use their argument. Both logics include constants and delta rules, and we prove their equivalence with the $\lambda_{\mathbf{v}}$-calculus.

1 Introduction

The translation of functional languages into combinatory logics captures the essence of compilation both from a theoretical as well as a practical perspective. Most of the research on this topic concerns the traditional call-by-name λ-calculus and combinatory logic. However, most realistic programming languages rely on call-by-value as the parameter-passing mechanism, for which a call-by-name combinatory logic (**CL**) is unsound. In addition, an evaluator for a language usually does not produce normal forms as in the λ-calculus [12], but instead produces values. For illustration of the mismatch between the $\lambda_{\mathbf{v}}$-calculus and **CL**, consider the following expression (in the $\lambda_{\mathbf{v}}$-calculus):

$$(\lambda xy.y)((\lambda w.ww)(\lambda w.ww))$$

This loops forever, while the corresponding combinatory logic expression:

$$(\mathsf{KI})((\mathsf{SII})(\mathsf{SII}))$$

reduces to I.

*Both authors were supported in part by NSF grant CCR 89-17022 and Darpa/NSF grant CCR 87-20277.

Moreover, unlike purely theoretical systems such as **CL**, programming languages also contain constants. Again, traditional theorems of equivalence between lambda theories and combinatory logics do not account for this fact, which means that such results are not really applicable to programming languages.

A solution to this problem is to translate by-value λ-terms with constants into a by-value combinatory logic with constants. Just as languages are paired with calculi [12], calculi should be paired with combinatory logics. In this paper, we show that a variant of Goodman's simplified combinatory logic [6] corresponds, in the sense of Curry [5], to Plotkin's $\lambda_{\mathbf{v}}$-calculus [12]. We also present a technique for determining if a combinatory logic and calculus are equivalent when both include constants. Unfortunately, the revised by-value combinatory logic leads to inefficient implementations, since the translation algorithm cannot preserve redexes that occur within abstractions. We solve this problem by introducing a modicum of laziness into the logic, together with new translation functions. This allows shorter combinator programs to be derived from $\lambda_{\mathbf{v}}$-calculus programs by preserving more of the original syntactic structure.

In section 2 we review the λ-calculus, **CL**, and the equivalence between them. Section 3 covers Plotkin's $\lambda_{\mathbf{v}}$-calculus. Section 4 presents a variant of Goodman's by-value combinatory logic, defines an abstraction algorithm and translations between the combinatory logic and $\lambda_{\mathbf{v}}$-calculus. Goodman's combinatory logic is proved equivalent to the $\lambda_{\mathbf{v}}$-calculus. Next, in section 5 we present an improved combinatory logic and prove its equivalence with the $\lambda_{\mathbf{v}}$-calculus. The conclusion is presented in section 6.

Notation & Conventions: Boldface letters, like **CL** or $\boldsymbol{\lambda}$, denote equational theories. Italic letters, like *CL*, and the greek letter Λ denote term languages. The notation FV denotes the free variables of a term or terms. Substitution is hygienic. That is, when substituting a term N for a variable x in a term M, we assume that the free variables of N and the bound variables of M are disjoint.

2 Background

Church [3] developed the λ-calculus [1]. Its term language Λ is:

$$M ::= x \mid \lambda x.M \mid MM$$

and the axiom and inference rule is:

$$\begin{aligned} (\lambda x.M)N &= M[x/N] && \beta \\ M = N &\Rightarrow \lambda x.M = \lambda x.N && \xi \end{aligned}$$

where $M[x/N]$ denotes substitution. The equational theory $\boldsymbol{\lambda}$ is generated from β and the usual congruence axioms/inference rules of Figure 1, and we write $\boldsymbol{\lambda} \vdash M = N$ when an equation is valid in $\boldsymbol{\lambda}$.

$$
\begin{aligned}
M &= M && \text{(reflexivity)}\\
M = N &\Rightarrow N = M && \text{(symmetry)}\\
M = N, N = L &\Rightarrow M = L && \text{(transitivity)}\\
M = N &\Rightarrow ML = NL,\ LM = LN && \text{(compatibility)}
\end{aligned}
$$

FIGURE 1: These axioms and inference rules are present in all equational theories of this paper.

Schönfinkel [10] and Curry [4] independently developed combinatory logic, based on the combinators S and K. The term language CL is:

$$M ::= x \mid \mathsf{S} \mid \mathsf{K} \mid MM;$$

the axioms are:

$$
\begin{aligned}
\mathsf{K}MN &= M\\
\mathsf{S}LMN &= LN(MN).
\end{aligned}
$$

The equational theory generated, including the axioms and inference rules of figure 1, is denoted by **CL**. Curry [5] also developed functions, $[\cdot]_\lambda$ and $[\cdot]_{CL}$, that translate terms from CL to Λ and vice versa, respectively. However, **CL** is strictly weaker than $\boldsymbol{\lambda}$: some terms that are not equal in **CL** have translations that are equal in $\boldsymbol{\lambda}$. Curry proved the

$$
\begin{aligned}
\mathsf{K} &= \lambda^* xy.\mathsf{K}xy\\
\mathsf{S} &= \lambda^* xyz.\mathsf{S}xyz\\
\lambda^* xy.\mathsf{S}(\mathsf{K}x)(\mathsf{K}y) &= \lambda^* xy.\mathsf{K}(xy)\\
\lambda^* xy.\mathsf{S}(\mathsf{S}(\mathsf{KK})x)y &= \lambda^* xyz.xz\\
\lambda^* xyz.\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{KS})x)y)z &= \lambda^* xyz.\mathsf{S}(\mathsf{S}xz)(\mathsf{S}yz)
\end{aligned}
$$

FIGURE 2: The Axiom Set A_β

equivalence of **CL** and $\boldsymbol{\lambda}$ in three ways. First, he added five axioms, A_β (see Figure 2), to **CL** to provide the extra power needed and proved the equivalence.

Theorem 2.1 (Curry) *The λ-calculus and* $\mathbf{CL} + A_\beta$ *are equivalent in the following sense:*

1. $\boldsymbol{\lambda} \vdash [[M]_{CL}]_\lambda = M$
2. $\mathbf{CL} + A_\beta \vdash [[M]_\lambda]_{CL} = M$
3. $\boldsymbol{\lambda} \vdash M = N \Leftrightarrow \mathbf{CL} + A_\beta \vdash [M]_{CL} = [N]_{CL}$
4. $\mathbf{CL} + A_\beta \vdash M = N \Leftrightarrow \boldsymbol{\lambda} \vdash [M]_\lambda = [N]_\lambda$

Remark: We use the notation $T_1 \sim T_2$ to denote the correspondence, in the sense of Theorem 2.1, between a **CL**-theory T_1 and a $\boldsymbol{\lambda}$-theory T_2.

Second, Curry [5, pages 201—202] proved that a single inference rule, ζ', is equivalent to A_β. Let *Funs* be the syntactic category of functions; for the λ-calculus, functions are λ-terms, for **CL** they are terms of the form: $\mathsf{S}, \mathsf{K}, \mathsf{S}M, \mathsf{K}M, \mathsf{S}MN$. Then ζ' is:

$$\forall F_1, F_2 \in \mathit{Funs}, F_1 x = F_2 x \;\Rightarrow\; F_1 = F_2 \text{ if } x \notin FV(F_1, F_2) \qquad (\zeta')$$

This new inference rule formalizes the notion of extensionality for functions.

Theorem 2.2 (Curry) $\mathbf{CL} + \zeta' \sim \boldsymbol{\lambda}$

Adding ζ' to the λ-calculus does not change the theory, so in fact $\boldsymbol{\lambda} = \boldsymbol{\lambda} + \zeta'$. This is not true for the impure λ-calculus with constants. For example, the term $\lambda x.succ\ x$ is not provably equal to *succ* without using ζ'. In addition, the theories generated from the axiom set A_β and from ζ' are the same:

Theorem 2.3 (Curry) $\mathbf{CL} + A_\beta \vdash M = N \Leftrightarrow \mathbf{CL} + \zeta' \vdash M = N.$

Third and finally, extensionality is added to both **CL** and $\boldsymbol{\lambda}$ and then proved their equivalence. The infererence rule ζ captures a stronger notion of extensionality than ζ':

$$Mx = Nx \;\Rightarrow\; M = N \text{ if } x \notin FV(M, N) \qquad (\zeta)$$

where $FV(M, N)$ is the set of free variables in M and N. Unlike ζ', it applies to all terms instead of terms that are syntactically functions. In the presence of constants it is unsound[1].

Theorem 2.4 (Curry) $\mathbf{CL} + \zeta \sim \boldsymbol{\lambda} + \zeta.$

Of course, $\boldsymbol{\lambda} + \zeta$ is not the same equational theory as $\boldsymbol{\lambda}$.

---

[1]Which is why Dana Scott, Robert Cartwright, and others have argued that ζ is too strong and should not be called the axiom of extensionality.

3 The $\lambda_{\mathbf{v}}$-Calculus

Plotkin's $\lambda_{\mathbf{v}}$-calculus is a by-value variant of the traditional λ-calculus. In this section we describe its syntax and semantics. The calculus presented here is a slight variation of Plotkin's $\lambda_{\mathbf{v}}$-calculus: it has functions as a separate syntactic domain to facilitate the definition of an extensionality rule in the presence of constants.

Definition 3.1. $(\boldsymbol{\lambda}_v)$
Syntactic Domains:

$$\begin{array}{rcll} x & \in & \mathit{Vars} & \text{(variables)} \\ a & \in & \mathit{Consts} & \text{(constants)} \\ a_f & \in & \mathit{Fun\text{-}Consts} & \text{(function constants)} \\ V & \in & \mathit{Vals} & \text{(values)} \\ F & \in & \mathit{Funs} & \text{(functions)} \\ M, N & \in & \Lambda_v & (\Lambda_v \text{ terms}) \end{array}$$

Syntax:

$$\begin{array}{rcl} F & ::= & \lambda x.M \mid a_f \\ V & ::= & x \mid a \mid F \\ M & ::= & V \mid MM \end{array}$$

Axioms and Inference Rules:

$$\begin{array}{rcll} (\lambda x.M)V & = & M[x/V] & (\beta_v) \\ a_f V & = & \delta_\lambda(a_f, V) \text{ if } \delta_\lambda(a_f, V) \text{ is defined} & (\delta) \\ M = N & \Rightarrow & \lambda x.M = \lambda x.N & (\xi) \\ F_1 x = F_2 x & \Rightarrow & F_1 = F_2 \text{ if } x \notin FV(F_1, F_2) & (\zeta') \end{array}$$

As usual, the axioms and inference rules of figure 1 are included. We use the notation $M[x/N]$ to denote the result of substituting all free occurrences x in M by N. The set of free variables of a term M, $FV(M)$, and substitution are defined as usual [1]. When an equation $M = N$ is derivable in the $\lambda_{\mathbf{v}}$-calculus we write $\boldsymbol{\lambda}_v \vdash M = N$.

The delta axiom is used to provide meaning to terms of the form $a_f V$, such as $\mathit{succ}\ 5$. The ζ' inference rule provides extensionality, and is based on the syntactic category for functions. Functions are either λ-terms or functional constants. The calculus needs some form of extensionality, otherwise equations such as $\lambda x.\mathit{succ}\ x = \mathit{succ}$ are not provable. Full extensionality (ζ) is too strong; it is unsound with respect to operational equivalence [12] because it does not respect values. Even with ζ-value, $\lambda x.3x = 3$ because $(\lambda x.3x)x = 3x$ for all x. Then, if the $\lambda_{\mathbf{v}}$-calculus contains a constant function *integer?*, we can show that $(\mathit{integer?}\ (\lambda x.3x)) = \mathit{false}$, and $(\mathit{integer?}\ 3) = \mathit{true}$, resulting in an inconsistent theory. Thus, ζ' is the appropriate choice.

4 A Simple Combinatory Logic

The combinatory logic presented here is a variant of Goodman's [6]: we also include functional extensionality (ζ') and constants. We refer to it as $\mathbf{CL_v}$; it is based on the by-value combinators S, K and I[2].

Definition 4.1. **($\mathbf{CL_v}$)**
Syntactic Domains:

$$
\begin{array}{rcll}
x & \in & \mathit{Vars} & \text{(variables)} \\
a & \in & \mathit{Consts} & \text{(constants)} \\
a_f & \in & \mathit{Fun\text{-}Consts} & \text{(function constants)} \\
U, V, W & \in & \mathit{Vals} & \text{(values)} \\
F & \in & \mathit{Funs} & \text{(functions)} \\
M, N & \in & CL_v & (CL_v \text{ terms})
\end{array}
$$

Syntax:

$$
\begin{array}{rcl}
F & ::= & a_f \mid \mathsf{I} \mid \mathsf{S} \mid \mathsf{K} \mid \mathsf{S}V \mid \mathsf{S}VV \mid \mathsf{K}V \\
V & ::= & a \mid x \mid F \\
M & ::= & V \mid MM
\end{array}
$$

Axioms and Inference Rule:

$$
\begin{array}{rcll}
\mathsf{I}V & = & V & (I) \\
\mathsf{K}UV & = & U & (K) \\
\mathsf{S}UVW & = & UW(VW) & (S) \\
a_f V & = & \delta_{cl}(a_f, V) \text{ if } \delta_{cl}(a_f, V) \text{ is defined} & (\delta) \\
F_1 x = F_2 x & \Rightarrow & F_1 = F_2 \text{ if } x \notin FV(FF') & (\zeta')
\end{array}
$$

Again, the set of free variables of a term M: $FV(M)$, and substitution are defined as usual, and the axioms and inference rules include those of figure 1. When $\mathbf{CL_v}$ proves an equation $M = N$, we write $\mathbf{CL_v} \vdash M = N$.

Functions (F) are incomplete I, K, or S applications in the sense that they must be applied to at least one argument, possibly more, to match the left hand side of the I, K, or S axioms. For example, in order for the K axiom to be applied, K must appear with two arguments. Thus, the term K is still expecting arguments and is a function. The same is true of $\mathsf{K}V$ for arbitrary values V. In addition, function constants are also functions.

Values, as in the $\lambda_{\mathbf{v}}$-calculus, are either functions, constants or variables.

As with $\boldsymbol{\lambda}$ and $\mathbf{CL}$, the $\lambda_{\mathbf{v}}$-calculus is stronger than $\mathbf{CL_v}$. As noted before, the approach of adding full extensionality to both systems is unsound due to the presence of

---

[2]The combinator I can be replaced by the term SKK.

constants. A modification of Curry's first approach, namely extending $\mathbf{CL_v}$ with a set of axioms similar to A_β, yields a complex system with an infinite set of axioms. It is an open question whether there exists a finite set of axioms that would make $\mathbf{CL_v}$ equivalent to the $\lambda_\mathbf{v}$-calculus. Thus we are left with the second approach, using the inference rule ζ' to prove the equivalence of the two systems.

Next we develop translation functions for mapping terms from Λ_v to CL_v and vice versa. The translation functions must preserve functions and values, otherwise the problems presented in the introduction occur. The idea behind the translation is simple: Variables and constants are drawn from the same set, so they map to themselves. Application is present in both systems, so applications map to applications. Functions are slightly more difficult, they require an abstraction algorithm to create CL_v functions that have the same behavior as λ-terms. Finally, the combinators S, K, and I must be mapped to λ-terms that have the same behavior.

To simulate $\lambda x.M$, the abstraction algorithm constructs a combinator term that uses its argument in the same way that M uses x. Informally, the term $\lambda x.x$ takes an argument and returns it; this is what the I combinator does. The term $\lambda x.y$ takes an argument and ignores it, thus $\mathsf{K}y$ will have the same effect. Finally, the term $\lambda x.MN$ takes an argument, replaces all free occurrences of x in M and N with that argument, and applies the resulting terms. Another way of writing this is: $\lambda x.(\lambda x.M)x((\lambda x.N)x)$. So in this case, let M_1 and N_1 be the combinatory terms for $\lambda x.M$ and $\lambda x.N$ respectively, then using $\mathsf{S}M_1N_1$ will have the desired effect. This is the simplest algorithm for simulating abstraction as well as handling the issue of preserving values. The translation algorithm is called λ_1 following Barendregt [1, page 157].

Definition 4.2. (λ_1)
$\lambda_1 : \mathit{Vars} \times CL_v \to CL_v$ (simulating λ)

$$\begin{aligned}
\lambda_1 x.x &\equiv \mathsf{I} \\
\lambda_1 x.c &\equiv \mathsf{K}c \text{ if } c \not\equiv x \in \mathit{Vars} \text{ or } c \in \mathit{Consts} \text{ or } c \in \{\mathsf{S},\mathsf{K},\mathsf{I}\} \\
\lambda_1 x.(MN) &\equiv \mathsf{S}(\lambda_1 x.M)(\lambda_1 x.N)
\end{aligned}$$

The translation from the $\lambda_\mathbf{v}$-calculus to CL_v merely uses the abstraction algorithm λ_1 as needed.

Definition 4.3. ($[\cdot]_{CL}$)
$[\ ]_{CL} : \Lambda_v \to CL_v$

$$\begin{aligned}
[x]_{CL} &\equiv x \\
[a]_{CL} &\equiv a \\
[a_f]_{CL} &\equiv a_f \\
[\lambda x.M]_{CL} &\equiv \lambda_1 x.[M]_{CL} \\
[MN]_{CL} &\equiv [M]_{CL}[N]_{CL}
\end{aligned}$$

For the translation from CL_v to Λ_v, the combinators are mapped to appropriate λ-terms based on their axioms.

Definition 4.4. ($[\cdot]_\lambda$)
$[\ ]_\lambda : CL_v \to \Lambda_v$

$$
\begin{array}{rcl}
[x]_\lambda & \equiv & x \\
[a]_\lambda & \equiv & a \\
[a_f]_\lambda & \equiv & a_f \\
[\mathsf{I}]_\lambda & \equiv & \lambda x.x \\
[\mathsf{K}]_\lambda & \equiv & \lambda xy.x \\
[\mathsf{S}]_\lambda & \equiv & \lambda xyz.xz(yz) \\
[MN]_\lambda & \equiv & [M]_\lambda [N]_\lambda
\end{array}
$$

With this translation, some values (such as KK) do not map directly to values in the $\lambda_{\mathbf{v}}$-calculus, but instead map to terms that reduce to values.

With constants and delta rules in the two systems, we must have a criteria for determining if the two delta rules have the same behavior under the translation functions given above.

Definition 4.5. (*Compatible Delta Rules*)
Two delta rules, δ_λ and δ_{cl}, are *compatible* if

1. $\boldsymbol{\lambda}_v \vdash \delta_\lambda(a_f, V) = M \Rightarrow \mathbf{CL_v} \vdash \delta_{cl}([a_f]_{CL}, [V]_{CL}) = [M]_{CL}$
2. $\mathbf{CL_v} \vdash \delta_{cl}(a_f, V) = M \Rightarrow \boldsymbol{\lambda}_v \vdash \delta_\lambda([a_f]_\lambda, [V]_\lambda) = [M]_\lambda$

For all common δ_λ rules, there exist δ_{cl} rules that are compatible.

Theorem 4.6 $\boldsymbol{\lambda}_v \sim \mathbf{CL_v}$ *if* δ_λ *and* δ_{cl} *are compatible.*

Proof sketch. The proof follows the outline of Barendregt's Theorem 7.3.12. Two portions of the proof are presented here to demonstrate the use of functional extensionality.

Since one of the criteria for equivalence is that $\mathbf{CL_v} \vdash M = [[M]_\lambda]_{CL}$, we must show that $\mathbf{CL_v} \vdash \mathsf{K} = [[\mathsf{K}]_\lambda]_{CL}$.

$$
\begin{array}{rcl}
[[\mathsf{K}]_\lambda]_{CL} & \equiv & [\lambda xy.x]_{CL} \\
 & \equiv & \mathsf{S(KK)I}
\end{array}
$$

To show that $\mathbf{CL_v} \vdash \mathsf{K} = \mathsf{S(KK)I}$ we use the functional extensionality rule. To prove the antecedent, we calculate:

$$\mathsf{S(KK)I}x = (\mathsf{KK}x)(\mathsf{I}x) = \mathsf{K}(\mathsf{I}x) = \mathsf{K}x$$

and so by functional extensionality:

$$\mathsf{K} = \mathsf{S(KK)I}$$

A slightly more complicated example comes from the ξ inference rule: For it to hold true in $\mathbf{CL_v}$, we must show that $M = N \Rightarrow \lambda_1 x.M = \lambda_1 x.N$. This can be proved by induction on the length of proof of $M = N$. For example, consider the S axiom $\mathsf{S}UVW = UW(VW)$ used as the last step of the proof of $M = N$. We must show

$$\mathbf{CL_v} \vdash \lambda_1 x.\mathsf{S}UVW = \lambda_1 x.UW(VW)$$

First,

$$\lambda_1 x.\mathsf{S}UVW \equiv \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{KS})\lambda_1 x.U)\lambda_1 x.V)\lambda_1 x.W$$

and

$$\lambda_1 x.UW(VW) \equiv \mathsf{S}(\mathsf{S}(\lambda_1 x.U)(\lambda_1 x.W))(\mathsf{S}(\lambda_1 x.V)(\lambda_1 x.W))$$

Now consider applying both terms to x:

$$\begin{array}{rcl} \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{KS})\lambda_1 x.U)\lambda_1 x.V)(\lambda_1 x.W)x & = & \mathsf{S}UVW \\ & = & UW(VW) \\ \mathsf{S}(\mathsf{S}(\lambda_1 x.U)(\lambda_1 x.W))(\mathsf{S}(\lambda_1 x.V)(\lambda_1 x.W))x & = & UW(VW) \end{array}$$

Since they are equal for all x, ζ' applies and the S axiom is closed under λ_1. □

5 Introducing Laziness

The λ_1 abstraction algorithm does not preserve the structure of terms. Consider the program: $\lambda x.(\lambda y.y)x$. This program is translated into $\mathsf{S}(\mathsf{KI})\mathsf{I}$, which, unlike the λ-term, contains no redexes. In the by-name λ-calculus, this problem was solved by using λ^* as the abstraction algorithm. λ^* differs from λ_1 in one clause:

$$\lambda^* x.M \equiv \mathsf{K}M \text{ if } x \notin FV(M)$$

That is, λ^* produces $\mathsf{K}M$ where M is the largest possible term with x not occurring free. On the other hand, λ_1 will only produce $\mathsf{K}M$ if M is a constant, variable, or combinator. Using λ^* as the by-value abstraction algorithm causes some values in the $\lambda_\mathbf{v}$-calculus to map to non-value terms in CL_v. This will not work for the by-value case due to the restriction that values must be translated to values: $\lambda x.(\lambda y.yy)(\lambda y.yy)$ is a value in the $\lambda_\mathbf{v}$-calculus, but its translation $\mathsf{K}((\mathsf{SII})(\mathsf{SII}))$ is not a value in CL_v nor does it reduce to one. If the value set is extended so that $\mathsf{K}M$ is a value, then the $\lambda_\mathbf{v}$-calculus program:

$$(\lambda xy.x)((\lambda y.yy)(\lambda y.yy))$$

is translated to

$$(\mathsf{S}(\mathsf{KK})\mathsf{I})((\mathsf{SII})(\mathsf{SII})),$$

which, by functional extensionality, is

$$\mathsf{K}((\mathsf{SII})(\mathsf{SII}))$$

This belongs to the extended set of values even though the λ-term does not.

The algorithm λ^* can be modified to work with the $\lambda_\mathbf{v}$-calculus by restricting M in the above clause to a value:

$$\lambda^* x.V \equiv \mathsf{K}V \text{ if } x \notin FV(V)$$

While this is correct, it does not achieve the desired result. Values in CL_v contain no redexes, so λ^* stills splits redexes (as in the λ_1 algorithm).

To avoid splitting redexes, we modify the combinatory logic by replacing the combinator K with the syntax $(\mathsf{K}_l M)$. A K_l term suspends the evaluation of its subexpression. A term $(\mathsf{K}_l M)$ is a function (and thus a value), and accepts an argument by-value, as usual. To avoid the problem of values not translating to values, the term language is restricted so that K_l always appears with its one sub-term, i.e., K_l is not a combinator. To emphasize this, we always write K_l terms with explicit parenthesis.

Abstracting x from the term $(\mathsf{K}_l x)$ cannot be $\mathsf{S}(\mathsf{K}_l\mathsf{K}_l)\mathsf{I}$ since this is not a valid expression. To handle this problem the combinator Q is added to define abstraction over K_l terms[3]. This combinatory logic is called $\mathbf{CL_q}$.

Definition 5.1. $(\mathbf{CL_q})$

Syntactic Domains are the same as for $\mathbf{CL_v}$.

The syntax is similar to $\mathbf{CL_v}$ except for the restriction on K_l and the addition of Q.

Syntax:

$$\begin{array}{rcl} F & ::= & a_f \mid \mathsf{I} \mid \mathsf{S} \mid \mathsf{Q} \mid (\mathsf{K}_l M) \mid \mathsf{S}V \mid \mathsf{S}VV \mid \mathsf{Q}V \\ V & ::= & a \mid x \mid F \\ M & ::= & V \mid MM \end{array}$$

Axioms and Inference Rule:

$$\begin{array}{rcll} \mathsf{I}V & = & V & (I) \\ (\mathsf{K}_l M)V & = & M & (K_l) \\ (\mathsf{Q}UV) & = & (\mathsf{K}_l(UV)) & (Q) \\ \mathsf{S}UVW & = & UW(VW) & (S) \\ a_f V & = & \delta_{cl}(a_f, V) \text{ if } \delta_{cl}(a_f, V) \text{ is defined} & (\delta) \\ F_1 x = F_2 x & \Rightarrow & F_1 = F_2 \text{ if } x \notin FV(FF') & (\zeta') \\ M = N & \Rightarrow & (\mathsf{K}_l M) = (\mathsf{K}_l N) & (\xi_k) \end{array}$$

Again, the axioms and inference rules of figure 1 are included. The new abstraction algorithm, $\lambda^\sharp$, is similar to the λ_1 algorithm except in the details mentioned above, and the problem that abstraction over K_l terms is not defined. Consider a term $\lambda^\sharp x.(\mathsf{K}_l M)$ with x free in M. Performing abstraction as in the λ_1 algorithm would produce an illegal term: $\mathsf{S}(\mathsf{K}_l\mathsf{K}_l)(\lambda^\sharp x.M)$. Instead, we need a term that takes an argument, passes it to K_l's first sub-term, and then throws away the next argument it receives. This is exactly what the Q combinator does.

In addition, functional extensionality is incorporated into the algorithm: terms of the form $\lambda^\sharp x.Fx$ are replaced with F, since this aids in producing shorter terms.

[3]The Q combinator is not related to that appearing in Smullyan [13].

Definition 5.2. ($\lambda^\sharp$)
$\lambda^\sharp : \mathit{Vars} \times CL_q \to CL_q$

$$\begin{array}{rcl}
\lambda^\sharp x.x & \equiv & \mathsf{I} \\
\lambda^\sharp x.Fx & \equiv & F \text{ if } x \notin FV(F) \\
\lambda^\sharp x.M & \equiv & (\mathsf{K}_l M) \text{ if } x \notin FV(M) \\
\lambda^\sharp x.(\mathsf{K}_l M) & \equiv & \mathsf{Q}(\lambda^\sharp x.M) \\
\lambda^\sharp x.(MN) & \equiv & \mathsf{S}(\lambda^\sharp x.M)(\lambda^\sharp x.N)
\end{array}$$

The translation function from Λ_v to CL_q is the same as for Λ_v to CL_v, except that λ_1 is replaced by $\lambda^\sharp$. Translating terms from CL_q to Λ_v proceeds as before except for K_l terms and Q combinators. Q's definition is added — it has a simple translation to a λ-term.

Definition 5.3. ($[\cdot]_\lambda$)
$[\ ]_\lambda : CL_q \to \Lambda_v$

$$\begin{array}{rcl}
[x]_\lambda & \equiv & x \\
[a]_\lambda & \equiv & a \\
[a_f]_\lambda & \equiv & a_f \\
[\mathsf{I}]_\lambda & \equiv & \lambda x.x \\
[(\mathsf{K}_l M)]_\lambda & \equiv & \lambda x.[M]_\lambda,\ x \notin FV(M) \\
[\mathsf{Q}]_\lambda & \equiv & \lambda xy.\lambda z.xy \\
[\mathsf{S}]_\lambda & \equiv & \lambda xyz.xz(yz) \\
[MN]_\lambda & \equiv & [M]_\lambda[N]_\lambda
\end{array}$$

Now the $\lambda_{\mathbf{v}}$-calculus and $\mathbf{CL_q}$ are equivalent.

Theorem 5.4 $\boldsymbol{\lambda}_v \sim \mathbf{CL_q}$ *if* δ_λ *and* δ_{cl} *are compatible.*

The proof is similar to that of Theorem 4.6.

The system $\mathbf{CL_q}$ has several advantages over $\mathbf{CL_v}$. Consider the $\lambda_{\mathbf{v}}$-calculus program $\lambda x.\mathit{succ}\ 2$. The translation to CL_v produces $\mathsf{S}(\mathsf{K}\ \mathit{succ})(\mathsf{K}2)$ while the CL_q version is $(\mathsf{K}_l(\mathit{succ}\ 2))$. Preservation of such redexes permits the use of conventional compiler optimizations on the terms, in this case, the redex can be replaced with 3. The CL_q program is shorter, requiring a single combinator as opposed to three. Indeed, the translation of any program to CL_q produces a shorter term than the translation to CL_v.

The CL_q translation also has the property that it produces maximally free expressions. Given a λ-term $\lambda x.M$, the maximally free expressions are those subterms of M in which x does not occur free, and are not contained in any other maximally free expression. The maximally free expressions are the terms that occur as the first argument of a K_l term.

Maximally free expressions provide a technique for ensuring fully lazy evaluation. An implementation is fully lazy when a combinator term that does not depend on its argument is evaluated only once, no matter how many times it is shared through the program. For example, given the program $\mathsf{SII}(\mathsf{K}_l(\mathsf{II}))$, an implementation only needs to evaluate the (II) redex once, even though the first reduction step causes it to be duplicated: $\mathsf{I}(\mathsf{K}_l(\mathsf{II}))(\mathsf{I}(\mathsf{K}_l(\mathsf{II})))$, which in a graph reduction implementation means it will be shared. $\mathbf{CL_q}$ can be implemented so that evaluation of the two redexes will be shared.

6 Conclusion

The two combinatory logics, $\mathbf{CL_v}$ and $\mathbf{CL_q}$, presented in this paper are interesting for several reasons. The simple logic, $\mathbf{CL_v}$, may be useful for developing an algebraic model for the $\lambda_{\mathbf{v}}$-calculus, as **CL** was used to create a model of the λ-calculus [11]. An open problem is the construction of a set of axioms, corresponding to A_β, for $\mathbf{CL_v}$ that are equivalent to the inference rule ζ'.

The logic $\mathbf{CL_q}$ is useful for creating implementations of by-value languages. Burge [2] rediscovered combinators for computer science. Turner [14] popularized and improved the translation using a larger set of combinators so that the resulting **CL** programs were shorter. Then Hughes [8] developed supercombinators that made the translation linear in terms of size. Hudak and Goldberg [7] created serial combinators that are used to execute **CL** programs on multiprocessors without shared memory. Finally, Kennaway and Sleep [9] developed director strings that allow even more efficient implementations. Just as **CL** has been improved in various ways, $\mathbf{CL_q}$ is the appropriate starting point for a parallel line of research for by-value languages.

7 Acknowledgements

We would like to thank our families for their support. We give sincere thanks to Matthias Felleisen and Dan Friedman for their guidance and patience with this paper. Also, we wish to thank Mike Fagan, Amr Sabry, and Andrew Wright for their careful proofreading.

References

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, revised edition, 1984.

[2] W. Burge. *Recursive Programming Techniques*. Addison-Wesley, Reading, Mass., 1975.

[3] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, 1941.

[4] H. B. Curry. Grundlagen der kombinatorischen Logik. *Amer. J. Math*, 52:509–536; 789–834, 1930.

[5] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[6] N. D. Goodman. A simplification of combinatory logic. *Journal of Symbolic Logic*, 37(2), 1972.

[7] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In *Proc of Conf. on Functional Prog. Langs. and Comp. Arch.*, 1985.

[8] R. Hughes. Super-combinators: A new implementation method for applicative languages. In *Sym. on Lisp and Functional Prog.*, pages 1–10. ACM, Aug 1982.

[9] Kennaway, R. and R. Sleep. Director strings as combinators. *ACM Trans. Prog. Lang. Syst.*, 10(4):602–626, 1988.

[10] M. Schönfinkel. Über die Bausteine der Mathematischen Logik. *Math. Annalen*, 92:305–316, 1924.

[11] A. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[12] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[13] R. Smullyan. *To Mock a Mockingbird and Other Logic Puzzles.* Alfred A. Knopf, Inc., New York, 1985.

[14] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.