On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis*

John Field†
Cornell University

Abstract

In this paper, I introduce a new formal system, **ACCL**, based on Curien's *Categorical Combinators* [Cur86a]. I show that **ACCL** has properties not possessed by Curien's original combinators that make it particularly appropriate as the basis for implementation and analysis of a wide range of reduction schemes using shared environments, closures, or λ-terms. As an example of the practical utility of this formalism, I use it to specify a simple lazy interpreter for the λ-calculus, whose correctness follows trivially from the properties of **ACCL**.

I then describe a *labeled* variant of **ACCL**, $\mathbf{ACCL}^{\mathbf{L}}$, which can be used as a tool to determine the degree of "laziness" possessed by various λ-reduction schemes. In particular, $\mathbf{ACCL}^{\mathbf{L}}$ is applied to the problem of *optimal* reduction in the λ-calculus. A reduction scheme for the λ-calculus is optimal if the number of redex contractions that must be performed in the course of reducing any λ-term to a normal form (if one exists) is guaranteed to be minimal. Results of Lévy [Lév78,Lév80] showed that for a natural class of reduction strategies allowing *shared* redexes, optimal reductions were, at least in principle, possible. He conjectured that an optimal reduction strategy might be realized in practice using shared closures and environments as well as shared λ-terms. I show, however, using $\mathbf{ACCL}^{\mathbf{L}}$, a practical optimal reduction scheme for arbitrary λ-terms using only shared environments, closures, or terms is unlikely to exist.

1 Background

There has been much recent interest in efficient implementations of lazy functional programming languages whose semantics are based on normalizing reduction schemes for the λ-calculus [Pey87,FH88]. Most such implementations have made use of some combination of the notions of *graph reduction* [Wad71,Aug84,Joh84], *environments* [Lan64,HM76, AP81,FW87] or *combinators* [Tur79,Hug84,Joh85]. The first two are means to allow certain redexes to be effectively shared during reduction; the latter can be considered a restricted form of λ-expression for which certain implementation techniques are more efficient.

While all these methods are *normalizing*, that is, guaranteed to yield a normal form[1] if one exists, all end up performing more β-contractions than are absolutely necessary by effectively *copying* redexes. In some cases, this lack of sufficient laziness can result in considerable unnecessary additional computation. Concern for this phenomenon led to the introduction of methods allowing "fully-lazy" reduction [Hug84]. However, J.-J. Lévy's analysis [Lév78,Lév80] made clear that there was a wide range of laziness possible, ranging from profligate (simple leftmost β-reduction without sharing) to optimal, with full-laziness actually somewhere in between. The exact nature of laziness in various implementation has apparently heretofore been something of a mystery[2], and I aim here to give means to analyze this phenomenon more precisely.

This paper presupposes a familiarity with the λ-calculus [Chu41,Bar84,HS86], the de Bruijn λ-calculus [dB72,dB78, Cur86a], and basic ideas from term rewriting systems [HO80, Hue80,Der87]. A brief review of relevant concepts and notation for these subjects is provided in Appendix A. An acquaintance with with Curien's Categorical Combinators [Cur86a,Cur86b,CCM87], and with the work of Lévy on optimality [Lév78,Lév80] would also be useful.

*This research was supported by NSF grant DCR 82-02677 and ONR grant N000014-88K-0594.

†Author's Address: Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853. Electronic mail: field@cs.cornell.edu.

[1]Technically, implementations of functional languages generally yield *weak head normal forms*.

[2]Peyton Jones [Pey87, p. 400] states that "...it is by no means obvious how lazy a function is, and...we do not at present have any tools for reasoning about this. Laziness is a delicate property of a function, and seemingly innocuous program transformations may lose laziness."

2 Redex Sharing and Environments

Consider the λ-term $M \equiv (\lambda y.(yy))(Iz)$, *where* $I \equiv \lambda x.x$. It may be reduced to a normal form in any one of three ways:

Example 2.1

$$\sigma_1: M \longrightarrow (Iz)(Iz) \longrightarrow z(Iz) \longrightarrow zz$$
$$\sigma_2: M \longrightarrow (Iz)(Iz) \longrightarrow (Iz)z \longrightarrow zz$$
$$\sigma_3: M \longrightarrow (\lambda y.(yy))z \longrightarrow zz$$

σ_1 is a *leftmost* reduction—one where the leftmost redex is contracted at each step. σ_3 is an *applicative order* reduction, where (informally) the argument part of a redex is reduced to a normal form before the redex is contracted. It is evident that σ_3 reaches the normal form (zz) in the fewest steps. It would clearly be desirable to have an *optimal* reduction strategy—one that always yields a normal form if one exists (i.e., is *normalizing*) and is also guaranteed to do so using the fewest possible redex contractions. Unfortunately, results of Barendregt, et al. [BBKV76], show that no such (recursive) strategy exists. However, we can improve matters considerably by extending the model of reduction a bit.

Note in the example above that the redex (Iz) of M is *copied* in reductions σ_1 and σ_2, since it is substituted for two instances of y. A natural alternative to copying expressions in arguments is to *share* them instead, using a graph-like data structure. The idea is illustrated below:

Example 2.2

$$\sigma_1': (\lambda y.(yy))(Iz) \longrightarrow (\bullet\bullet) \longrightarrow (\bullet\bullet) \equiv zz$$
$$\qquad\qquad\qquad\qquad\quad (Iz) \qquad\quad z$$

σ_1' proceeds from left to right, analogous to σ_1. In this case, however, the redex (Iz) is *shared*, rather than copied, as a result of its substitution for the two instances of variable y. The result of (Iz)'s reduction to z is shared as well. Using this method, the normal form's graph representation is reached after only two reduction steps.

Wadsworth's *graph reduction* algorithm [Wad71] formalizes the idea of Example 2.2. It combines a leftmost redex selection strategy with sharing of argument expressions. However, Wadsworth's algorithm is not *optimal*. If we contract non-leftmost redexes, shorter reductions (still using shared argument expressions) can be achieved, as the following example illustrates: Let $N \equiv (N_1 N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then the following are two graph reductions of N:

Example 2.3

$$\rho_1: N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (Iw)(\bullet z) \longrightarrow w(\bullet z)$$
$$\qquad\qquad\quad \lambda y.(Iy)\ \ \lambda y.(Iy) \qquad\quad \lambda y.(Iy)$$
$$\longrightarrow w(Iz) \longrightarrow wz$$

$$\rho_2: N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (\bullet w)(\bullet z) \longrightarrow w(\bullet z) \longrightarrow wz$$
$$\qquad\qquad\quad \lambda y.(Iy) \qquad\quad \lambda y.y \quad\ \lambda y.y$$

Wadsworth's algorithm performs reduction ρ_1, while ρ_2 reaches the normal form in fewer steps by contracting the shared (Iy) redex inside N_2 before applying it to either w or z (a minimal length reduction can also be achieved without any sharing by contracting the (Iy) redex before N_1 is applied to N_2).

Reducing inner redexes, as in ρ_2, seems to bring about shorter reductions in many cases. Unfortunately, contraction of arbitrary inner redexes can sometimes lead to unnecessarily diverging reductions, as is the case with the applicative order strategy. Wadsworth's scheme reduces only leftmost redexes in order to ensure normalizability (although this is not by any means the only way to do so, see [BKKS87]).

There is evidently a subtle interplay among the issues of efficiency, normalizability, and redex sharing. The quandary is then to find a way to edge closer to the brink of optimality without plunging into the abyss of non-normalizability.

By examining the reductions above, however, we can see that Wadsworth left the door open to further improvements by not taking advantage of all conceivable opportunities for redex sharing. Note in ρ_1 that as N_2 is applied in sequence to w and to z, the inner redex (Iy) is effectively copied (after each substitution for y). If there were some means to *parametrically* share the (Iy) redex while still substituting w and z separately for y, more efficient, and perhaps optimal reductions might still be achievable. This suggests the use of the notions of *environment* and *closure* familiar from implementations of programming languages.

2.1 Reduction Using Environments

A number of reduction schemes for the λ-calculus have been proposed using environments. These include that of Landin [Lan64] using applicative order evaluation, and updated versions devised by Henderson and Morris [HM76] and Aiello and Prini [AP81] to accommodate leftmost evaluation. Each of these systems avoids immediate substitutions for all instances of bound variables in the body of a λ-abstraction after β-contraction, constructing a closure instead.

To be more specific, an environment consists of sets of mappings between variable names and values, or *bindings*. The result of a β-contraction is then a *closure* consisting of the body of the abstraction part of the redex, paired with an environment updated to contain the binding of the abstraction's bound variable to the argument of the redex. The idea is illustrated below:

$$(\lambda x.(xx))N \longrightarrow [(xx),\ \langle\langle x := N \rangle\rangle]$$

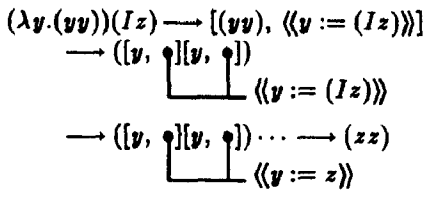
In general, $[T, E]$ will represent a closure consisting of term T and environment E. An environment is denoted thus:

$$\langle\langle B_1, B_2, \ldots \rangle\rangle$$

where B_1, B_2, etc. are bindings.

The following example (using the same term as in Example 2.2) shows that sharing of λ-terms can be achieved indirectly through shared bindings:

Example 2.4



Use of closures obviates copying any part of the body of an abstraction after β-contraction. Wadsworth's scheme, however, copies the parts of the body of an abstraction containing the abstraction's bound variable, in order to avoid incorrect substitutions in pieces of the abstraction's body that might be shared by other terms. By using environments, the body of the abstraction term, and hence any redexes contained therein, have the potential to be shared, avoiding redundant reductions.

Below is another reduction using the term of Example 2.3, showing that shared environments can be used to minimize the number of redex contractions performed in a nominally leftmost strategy: Once again, let $N \equiv (N_1 N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then, using shared environments, we have: (repeated meta-variables such as $\hat{E}$ below correspond to terms or environments shared through graphical data structures)

Example 2.5

$$
\begin{aligned}
N &\longrightarrow [(xw)(xz),\ \langle\!\langle x := N_2 \rangle\!\rangle] \longrightarrow \cdots \\
&\longrightarrow (([x,\ \hat{E}])[w,\ \hat{E}])\ [(xz),\ \hat{E}]), \\
&\qquad \textit{where} \quad \hat{E} \equiv \langle\!\langle x := N_2 \rangle\!\rangle \\
&\longrightarrow ((\hat{N}_2\ [w,\ \hat{E}])\ [(xz),\ \hat{E}], \\
&\qquad \textit{where} \quad \hat{E} \equiv \langle\!\langle x := \hat{N}_2 \rangle\!\rangle,\ \hat{N}_2 \equiv \lambda y.(Iy) \\
&\longrightarrow ([\hat{P},\ \langle\!\langle y := [w,\ \hat{E}] \rangle\!\rangle]\ [(xz),\ \hat{E}]), \\
&\qquad \textit{where} \quad \hat{E} \equiv \langle\!\langle x := \lambda y.\hat{P} \rangle\!\rangle,\ \hat{P} \equiv (Iy) \\
&\longrightarrow ([\hat{P},\ \langle\!\langle y := [w,\ \hat{E}] \rangle\!\rangle]\ [(xz),\ \hat{E}]), \\
&\qquad \textit{where} \quad \hat{E} \equiv \langle\!\langle x := \lambda y.\hat{P} \rangle\!\rangle,\ \hat{P} \equiv y \\
&\longrightarrow \cdots
\end{aligned}
$$

Note that the (Iy) redex in N_2 is reduced in a shared environment, *independently* of the substitution for free variable y in closures that refer to N_2

The question then arises as to whether some *combination* of shared environments, closures, and terms could be used to achieve an optimal reduction scheme, or at least improve on Wadsworth's method. To proceed any further, we will need a more formal system to study reduction using environments and closures.

3 ACCL

In [Cur86a], P.-L. Curien defines a number of equational theories based on Cartesian Closed Categories (CCCs) using terms from the *Pure Categorical Combinatory Logic*, **CCL**. Curien observed that the CCC axioms could model *reduction* in the λ-calculus, i.e., its operational semantics as well as its denotational semantics. Treated as combinators, Curien's axioms have the advantage of avoiding the difficulties with variables and substitution normally encountered in the λ-calculus, and thus has aspects in common with the *de Bruijn* λ-*calculus* [dB72,dB78].

One set of equational axioms, deemed *Weak Categorical Combinatory Logic*, is the basis for the Categorical Abstract Machine ([CCM87]). However, Curien proposed no system strong enough to simulate arbitrary β-reductions in the λ-calculus that could itself be simulated using only β-reduction. If such a system were available, it would provide an immediate proof of correctness for any reduction scheme for the λ-calculus based on it (since any combinator reduction would correspond to a β-reduction). λ-reduction methods based on Categorical Combinators proposed thus far, such as the Categorical Abstract Machine and schemes by Lins [Lin87], have heretofore required ad-hoc proofs of correctness.

To provide a more sophisticated tool for modeling λ-reduction using environments, I will define a new 2-sorted equational theory, **ACCL**, akin to Curien's theory **CCL**β. Its sort structure makes possible proofs of close correspondence between β-reduction and **ACCL** reduction not possible in Curien's original theory. While this modified term structure obscures the elegant categorical origins of Curien's original system, it makes its connection to reduction with environments much more evident.

In the sequel, I will assume that any λ-terms under consideration are actually terms of the de Bruijn λ-calculus, although I will feel free to give examples using named variables.

3.1 Term Structure

Definition 3.1 *The terms of* **ACCL** *are built from a set of variables and constructors over a two-sorted signature. The sorts are as follows:*

- $\mathcal{L}$, *the sort of* lambda-like expressions
- $\mathcal{E}$, *the sort of* environments

The constructors are listed below. Each constructor is given with the sort of the term constructed and the sorts of its argument(s) specified in the corresponding argument positions.

$$
\begin{aligned}
\mathrm{Var} &: \mathcal{L} && \textit{(variable reference)} \\
\mathrm{Apply}(\mathcal{L},\ \mathcal{L}) &: \mathcal{L} && \textit{(application)} \\
\Lambda(\mathcal{L}) &: \mathcal{L} && \textit{(abstraction)} \\
[\mathcal{L},\ \mathcal{E}] &: \mathcal{L} && \textit{(closure)} \\
\emptyset &: \mathcal{E} && \textit{(null environment)} \\
\Box &: \mathcal{E} && \textit{(shift)} \\
\langle \mathcal{E},\ \mathcal{L} \rangle &: \mathcal{E} && \textit{(expression list)} \\
\mathcal{E} \circ \mathcal{E} &: \mathcal{E} && \textit{(environment composition)}
\end{aligned}
$$

The terms of **ACCL** will be denoted by $Ter(\mathbf{ACCL})$ and the *closed terms*, those terms containing no variables, by $Ter_C(\mathbf{ACCL})$.

The following notation (for "de Bruijn" numbers) will be used:

Definition 3.2

$$
n! \equiv \begin{cases} \mathrm{Var} & n = 0 \\ [\mathrm{Var},\ \Box^n] & n > 0 \end{cases}
$$

where

$$\Box^n \equiv \begin{cases} \Box & n = 1 \\ \underbrace{\Box \circ (\Box \circ (\cdots (\Box \circ \Box) \cdots))}_{n \text{ times}} & n > 1 \end{cases}$$

The intuition behind the term structure of **ACCL** is fairly straightforward: Terms of sort $\mathcal{L}$ are analogous to terms in the de Bruijn λ-calculus, after variable numbers are encoded as above. Closures are created by the **ACCL** equivalent of β-contraction. Environments are essentially lists of terms, the association between bound variables and the terms to which they are bound being represented implicitly by position in the list. An environment informally presented as

$$\langle\!\langle x_1 := M_1, x_2 := M_2, \ldots, x_n := M_n \rangle\!\rangle$$

is represented in **ACCL** as

$$\langle\langle\langle \cdots \langle \emptyset,\ M_n\rangle \cdots\rangle,\ M_2\rangle,\ M_1\rangle.$$

"$\circ$" allows separate environments to be merged. The only perhaps mysterious term present is "$\Box$", which when composed on the left with an arbitrary environment effects the "shifting" of de Bruijn numbers required when environments are moved inside abstractions, and when composed on the right with an environment causes the outermost piece of the list to be stripped away in the course of variable lookup. All these operations are embodied in the axioms below:

3.2 Axioms

Definition 3.3 *The axioms of* **ACCL** *are as follows:*

(Beta)	$\mathrm{Apply}(\Lambda(A),\ B) = [A,\ \langle \emptyset,\ B\rangle]$
(AssC)	$[[A,\ E_1],\ E_2] = [A,\ E_1 \circ E_2]$
(NullEL)	$\emptyset \circ E = E$
(NullER)	$E \circ \emptyset = E$
(ShiftE)	$\Box \circ \langle E,\ A\rangle = E$
(VarRef)	$[\mathrm{Var},\ \langle E,\ A\rangle] = A$
(DA)	$[\Lambda(A),\ E] = \Lambda([A,\ \langle E \circ \Box,\ \mathrm{Var}\rangle])$
(DE)	$\langle E_1,\ A\rangle \circ E_2 = \langle E_1 \circ E_2,\ [A,\ E_2]\rangle$
(DApply)	$[\mathrm{Apply}(A,\ B),\ E] = \mathrm{Apply}([A,\ E],\ [B,\ E])$
(AssE)	$(E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3)$
(NullC)	$[A,\ \emptyset] = A$

I define a related equational theory, **ECCL**, as follows:

Definition 3.4 *The axioms of* **ECCL** *are those of* **ACCL** *without rule* **Beta**.

It will be useful to consider **ACCL** as the union of two systems intended for different purposes: **ECCL**, which governs manipulation of environments, and **(Beta)**, which models β-reduction.

3.3 ACCL as Rewriting System on Closed Terms

By orienting the equations of **ACCL** from left to right, they can be treated as a term rewriting system. The notation $\longrightarrow_{\mathrm{ACCL}}$ will be used to denote the application of a rule of **ACCL** in some context, i.e., $A \longrightarrow_{\mathrm{ACCL}} B$ if and only if $A \equiv C[X]$, X may be rewritten to Y using one of the oriented axioms of **ACCL**, and $B \equiv C[Y]$ (contexts are defined in Appendix A. I will use similar notation for **ECCL** and applications of single rules of **ACCL**, e.g. $\longrightarrow_{(\mathrm{Beta})}$, However, I will restrict myself in the sequel to the *closed terms* of **ACCL**, $Ter_C(\mathbf{ACCL})$. Since I am interested in using **ACCL** to model λ-reduction rather than to prove theorems, this restriction will be of no concern. More importantly, in conjunction with the 2-sorted term structure of **ACCL**, the restriction to closed terms makes it possible to prove properties of **ACCL** that did not hold for arbitrary terms of Curien's system **CCL**β. I will refer to the formal theories and their corresponding rewriting systems by the same name. The following properties hold of **ACCL**:

Theorem 3.1 **ECCL** *is noetherian (strongly normalizing).*

Proof We can orient the rules of **ECCL** by combining the *recursive path ordering* method of Dershowitz and the *lexicographic path ordering* method of Kamin and Lévy (both of which are described in [Der87]) using an extension of Lescanne's notion of *status* [Les84].

We first order the operators of **ACCL** as follows:

$$\emptyset < \Box < \mathrm{Var} < \Lambda(\cdot) < \mathrm{Apply}(\cdot,\ \cdot) < \langle \cdot,\ \cdot\rangle < [\cdot,\ \cdot] = \circ$$

Let A and B be terms of **ACCL**, whose outermost operators are f and g, respectively. We then define the following quasi-ordering such terms:

$$A \equiv f(s_1, \ldots, s_m) \succeq B \equiv g(t_1, \ldots, t_n)$$

if

$$s_i \succeq t, \quad \text{for some} \quad i = 1 \ldots m,$$

or

$$f > g \quad \text{and} \quad s \succ t_j \quad \text{for all} \quad j = 1 \ldots n,$$

or

$$f \equiv g, f \not\equiv \circ, f \not\equiv [\cdot,\ \cdot], \quad \text{and} \quad \{s_1, \ldots, s_m\} \succeq_M \{t_1, \ldots t_n\}$$

or

$$f \equiv g, f \equiv \circ \text{ or } f \equiv [\cdot,\ \cdot], \quad \text{and} \quad (s_1, \ldots, s_m) \succeq_* (t_1, \ldots t_n)$$

or

$$f = g, f \equiv \circ, g \equiv [\cdot,\ \cdot], \quad \text{and} \quad \{s_1, \ldots, s_m\} \succeq_M \{t_1, \ldots t_n\}$$

where $\succeq_M$ is the extension of $\succeq$ to multisets of terms and $\succeq_*$ is the lexicographic extension of $\succeq$ to sequences (see [Der87] for details of these extensions).

Depending on the "status" of *unordered pairs* of operators, either the multiset or lexicographic ordering is used to compare operands. The ordering defined above is a well-quasi-ordering on terms of **ACCL** since it meets Kamin and Lévy's requirements for a *simplification ordering* [KL80]. Generalizations of Lescanne's notion of status were suggested in [Rus87]. Using this ordering, it is straightforward to show that if $A \longrightarrow_{\mathrm{ECCL}} B$, $A \succ B$, and thus that **ECCL** is noetherian. $\Box$

Theorem 3.2 ECCL *is confluent (thus Church-Rossser) on closed terms, i.e.,*

$$A \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} B_1 \wedge A \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} B_2 \Longrightarrow (\exists C) \quad B_1 \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} C \wedge B_2 \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} C$$

Proof We can show **ECCL** confluent by showing critical pairs to be locally confluent [Hue80]. The only problem occurs with the rule pair **(DA)** and **(NullC)**, for which we must show

$$(\forall A)(\forall n) \quad [A, \langle\langle\langle\cdots\langle\Box^n, (n-1)!\rangle\cdots\rangle, 1!\rangle, 0!\rangle] \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} A$$

and

$$(\forall E)(\forall n) \quad E \circ \langle\langle\langle\cdots\langle\Box^n, (n-1)!\rangle\cdots\rangle, 1!\rangle, 0!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} E$$

which can be proved for closed terms by a straightforward induction on the structure of A or E. The 2-sorted structure of terms of **ACCL** is essential to this argument. □

We can also have the following

Theorem 3.3 *(***Beta***) is confluent, i.e.,*

$$A \longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})} B_1 \wedge A \longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})} B_2 \Longrightarrow (\exists C) \quad B_1 \longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})} C \wedge B_2 \longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})} C$$

Proof **(Beta)** redexes cannot overlap (i.e., there are no critical pairs), confluence thus follows trivially. □

We can now show **ACCL** confluent by a technique similar to the Tait/Martin-Löf proof of the Church-Rosser property for the λ-calculus. The following reduction relation will be useful:

Definition 3.5

$$\longrightarrow_{\mathbf{Dev}} \equiv \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}} \cdot \longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})} \cdot \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}}$$

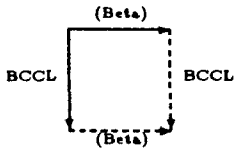
where '·' denotes relational composition.

$\longrightarrow_{\mathbf{Dev}}$ is intended to correspond roughly to the notion of a *development* in the λ-calculus. As usual, $\longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{Dev}}$ represents the reflexive, transitive closure of $\longrightarrow_{\mathbf{Dev}}$. I also define the following variant of **ECCL**:

Definition 3.6 *The axioms of* **BCCL** *consist of those of* **ECCL** *without rule (***DApply***).*

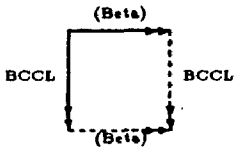
In order to show **ACCL** confluent, we need the following sequence of lemmas, each represented as a commuting diagram (dotted arrows denote reductions existentially dependent on the arbitrary reductions represented by solid arrows):

Lemma 3.1 BCCL *and (***Beta***) strongly commute, i.e.,*



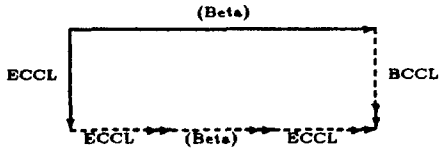
Proof Trivial, since **BCCL** and **(Beta)** have no critical pairs. □

Lemma 3.2 BCCL *and (***Beta***) commute, i.e.,*



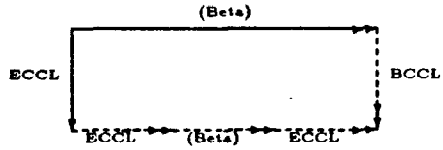
Proof Fill the diagram using lemma 3.1 (by induction on the lengths of the $\longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{BCCL}}$ and $\longrightarrow\!\!\!\!\!\rightarrow_{(\mathbf{Beta})}$ reductions). □

Lemma 3.3

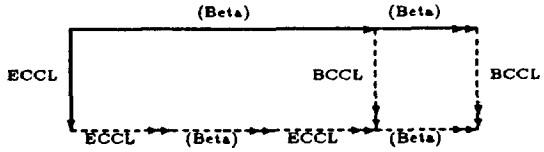


Proof One need only consider the critical pair of Apply([A(A), E], [B, E]) and [[A, ⟨∅, B⟩], E], for which it is easy to show there is a common reduct using the sort of reductions required by the lemma. □

Lemma 3.4

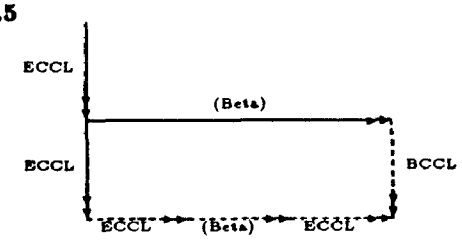


Proof If the **ECCL** rule used is not **(DApply)**, then the result follows from lemma 3.2. Otherwise, the **(DApply)** redex in the diagram's premise can also be a **(Beta)** redex. Without loss of generality, assume that some subterm is both a **(Beta)** redex and a **(DApply)** redex, and that it is the first redex contracted in the **(Beta)** reduction. (Since **(Beta)** redexes cannot create other **(Beta)** redexes, redexes in a **(Beta)** reduction can be permuted arbitrarily). We can then construct the desired diagram using lemmas 3.3 and 3.2 as follows:



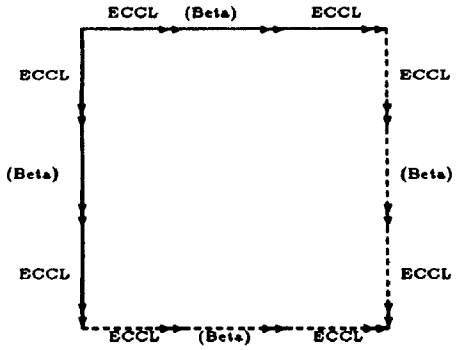
□

Lemma 3.5



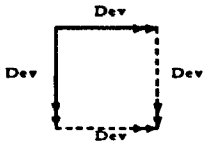
Proof Follows by noetherian induction (see [Hue80]) on the left-hand **ECCL** reduction using lemma 3.4 as a base case. (The rather odd $\longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ECCL}}$ appendage in the upper left-hand corner of the diagram is required to provide the appropriate induction hypothesis). □

Lemma 3.6



Proof Simple diagram construction using lemma 3.5, theorem 3.2, and theorem 3.3. □

Lemma 3.7 $\longrightarrow_{\text{Dev}}$ *is confluent on closed terms, i.e.,*



Proof The reductions used in lemma 3.6 are $\longrightarrow_{\text{Dev}}$ contractions, and the theorem thus follows by diagram chase. □

Theorem 3.4 ACCL *is confluent on closed terms.*

Proof $\longrightarrow\!\!\!\!\rightarrow_{\text{Dev}}$ and $\longrightarrow\!\!\!\!\rightarrow_{\text{ACCL}}$ are relationally equivalent. Thus from Lemma 3.7, we must conclude that $\longrightarrow\!\!\!\!\rightarrow_{\text{ACCL}}$ is confluent. □

Theorem 3.4 is a principal result; Curien was unable to exhibit a confluent system strong enough to model arbitrary reductions in the λ-calculus. However, independent work of Hardin [Har87,Har89] and Yokouchi [Yok89] has led to a characterization of *subsets* of Curien's original **CCL** terms for which confluence of the system **CCL**β can be proven. By contrast, the 2-sorted term structure of **ACCL** rules out the construction of "uninteresting" terms that Hardin and Yokouchi's **CCL** subsets explicitly omit.

Yokouchi's technique for proving the confluence of **CCL**β on subsets of terms is quite similar to the confluence proof given here. Lemma 3.5 was used in an earlier version of this paper to prove a somewhat stronger intermediate result than lemma 3.6; the proof used here was simplified upon observing that Yokouchi's proof of confluence essentially used lemma 3.5 directly, without resort to a more complicated intermediate lemma. Hardin's proof of relies on confluence of the λ-calculus.

Hardin and Yokouchi's proofs of confluence both rely on the fact that a "substitutive" subset of **CCL** similar to **ECCL** is noetherian. This was shown to be the case by Hardin and Laville [HL86], but required considerable ingenuity, since the substitutive part of **CCL** is apparently immune to more conventional techniques used to show termination. The proof that **ECCL** is noetherian is considerably simplified by its term structure, which in particular admits a distinction between closures and environment not present in **CCL**.

3.4 Normal Forms

Definition 3.7 *The set of* lambda normal forms *(LNF) is a subset of the terms of* **ACCL**, *defined inductively as follows:*

$$\begin{aligned} n! &\in LNF \\ A \in LNF &\implies \Lambda(A) \in LNF \\ A \in LNF, B \in LNF &\implies \text{Apply}(A,\ B) \in LNF \end{aligned}$$

Lambda normal forms are intuitively those terms that "look like" terms of the (de Bruijn) λ-calculus.

Theorem 3.5 *All lambda-like expressions (terms of sort $\mathcal{L}$) of* **ACCL** *are reducible to a lambda normal form, using the rules of* **ECCL**. *That is,*

$$(\forall A{:}\mathcal{L})\quad (\exists B \in LNF)\ s.t.\ A \longrightarrow\!\!\!\!\rightarrow_{\text{ECCL}} B$$

Proof Simply note that any term of sort $\mathcal{L}$ that is not in *LNF* contains an **ECCL** redex. Keep reducing such redexes using rules of **ECCL** until *LNF*is reached, which must happen eventually since **ECCL** is noetherian. □

For any term $A{:}\mathcal{L}$, I will refer to its corresponding term $B \in LNF$ by $lnf(B)$. Since **ECCL** is confluent and terms in *LNF* are irreducible in **ECCL**, this normal form is unique.

Definition 3.8 *The set of* partial environment normal forms *(PENF) is a subset of the terms of* **ACCL** *defined inductively as follows:*

$$\begin{aligned} \emptyset &\in PENF \\ \Box^n &\in PENF \\ E \in PENF &\implies \langle E,\ A\rangle \in PENF \end{aligned}$$

Theorem 3.6 *All environments (terms of sort $\mathcal{E}$) of* **ACCL** *are reducible to a partial environment normal form using the rules of* **ECCL**:

$$(\forall E_1{:}\mathcal{E})\quad (\exists E_2 \in PENF)\ s.t.\ E_1 \longrightarrow\!\!\!\!\rightarrow_{\text{ECCL}} E_2$$

Proof Once again, we can observe that every term of sort $\mathcal{E}$ that is not in *PENF* must contain an **ECCL** redex. Such redexes can be reduced until the normal form is reached. □

Terms in *PENF* are not necessarily irreducible in **ECCL**, thus partial environment normal forms are *not* unique.

3.5 Translation

We can now show state the translation between terms of the de Bruijn λ-calculus and terms of **ACCL**.

Definition 3.9 *For any term $M \in \lambda^{\text{DB}}$, we can define a corresponding term $[\![M]\!]_{\text{ACCL}} \in$ **ACCL** inductively as follows:*

$$\begin{aligned} [\![i]\!]_{\text{ACCL}} &= i! \\ [\![(\lambda.N)]\!]_{\text{ACCL}} &= \Lambda([\![N]\!]_{\text{ACCL}}) \\ [\![(N_1N_2)]\!]_{\text{ACCL}} &= \text{Apply}([\![N_1]\!]_{\text{ACCL}},\ [\![N_2]\!]_{\text{ACCL}}) \end{aligned}$$

The reverse transformation, $[\![\cdot]\!]_\lambda$, is defined in the obvious way on members of *LNF*.

3.6 Equivalence

I now claim that there is an equivalence between β-reduction and reduction of terms of sort $\mathcal{L}$ in **ACCL**. The following two lemmas are required:

Lemma 3.8 *Let M and N be arbitrary terms of the (de Bruijn) λ-calculus such that $M \longrightarrow_\beta N$. Then $[\![M]\!]_{\mathrm{ACCL}} \in LNF \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ACCL}} [\![N]\!]_{\mathrm{ACCL}} \in LNF$*

Proof A construction isomorphic to that used by Curien in [Cur86a] to prove a similar result for **CCL**β suffices, and is omitted here. The **ACCL** equivalent of his construction has the following property:

$$M \longrightarrow_\beta N \Longrightarrow (\exists B) \quad [\![M]\!]_{\mathrm{ACCL}} \longrightarrow_{(\mathbf{Beta})} B \quad \text{and} \quad B \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ECCL}} [\![N]\!]_{\mathrm{ACCL}}$$

□

Curien's construction yields the following corollary:

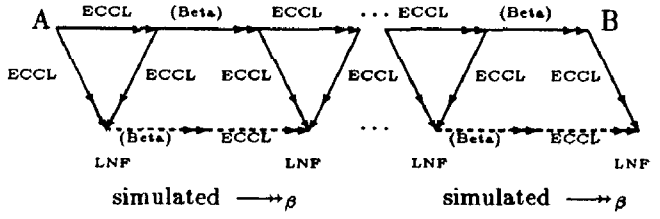
Corollary 3.1 *If $A \in LNF$, $C \in LNF$, and there exists B such that $A \longrightarrow\!\!\!\!\rightarrow_{(\mathbf{Beta})} B$ and $B \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ECCL}} C$ then $[\![A]\!]_\lambda \longrightarrow\!\!\!\!\rightarrow_\beta [\![B]\!]_\lambda$*

Proof Since (**Beta**) redexes are non-overlapping, we can perform Curien's β-simulation separately on each (**Beta**) redex contracted in the reduction from A to B, yielding a term in LNF at each stage. Once this process is complete, the resulting term must be C, since **ACCL** is confluent and $C \in LNF$. □

We can now prove the other direction:

Lemma 3.9 *Let $A: \mathcal{L} \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ACCL}} B$. Let $Inf(A) = A'$ and $Inf(B) = B'$. Then $[\![A']\!]_\lambda \longrightarrow\!\!\!\!\rightarrow_\beta [\![B']\!]_\lambda$.*

Proof Divide the **ACCL** reduction into subreductions alternating use of **ECCL** rules and uses of rule (**Beta**). The proof then reduces to showing that if $A_i \longrightarrow\!\!\!\!\rightarrow_{(\mathbf{Beta})} A_{i+1}$, $Inf(A_i) = A_i'$, $Inf(A_{i+1}) = A_{i+1}'$, then $[\![A_i']\!]_\lambda \longrightarrow\!\!\!\!\rightarrow_\beta [\![A_{i+1}']\!]_\lambda$. This can be done using corollary 3.1, which is used in the construction below:



□

We can use the construction of the term above to make the following definition:

Definition 3.10 *Let A be a term of* **ACCL** *containing a* (**Beta**) *redex B. Then the* residuals *of B (relative to the reduction of A to LNF) are those* (**Beta**) *redexes contracted in the proof of lemma 3.9 to simulate β-reduction in $Inf(A)$. The set of such residuals is denoted by $Resid(B, A)$.*

Putting the results from lemmas 3.8 and 3.9 together yields:

Theorem 3.7 *Given $M \in Ter(\lambda^{\mathrm{DB}})$,*

$$M \longrightarrow\!\!\!\!\rightarrow_\beta N \iff [\![M]\!]_{\mathrm{ACCL}} \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ACCL}} [\![N]\!]_{\mathrm{ACCL}}$$

This result shows that *any* reduction of a **ACCL** term $A \in LNF$ simulates a reduction in the λ-calculus.

We can now show that in terms of the number of (**Beta**) contractions performed, **ACCL** is always at least as efficient as the corresponding reduction in the λ-calculus:

Theorem 3.8 *Let $\sigma: A \longrightarrow\!\!\!\!\rightarrow_{\mathrm{ACCL}} B$ be a reduction in* **ACCL**. *Let $Inf(A) = A'$ and $Inf(B) = B'$. Let $\rho: [\![A']\!]_\lambda \longrightarrow\!\!\!\!\rightarrow_\beta [\![B']\!]_\lambda$ be the reduction given by Lemma 3.9. Then the number of β-contractions in ρ is greater than or equal to the number of* (**Beta**) *contractions in σ.*

Proof Direct corollary of proof of Lemma 3.9. □

Any reduction scheme for the λ-calculus implemented using **ACCL** would have to perform **ECCL** reductions as well as (**Beta**) contractions, but it is not unreasonable to count the former as "overhead," as do many other reduction schemes that manipulate environments as well as contracting β-redexes. One can generally show that in a reasonable reduction scheme, the number of **ECCL** reductions required is proportional to the number of (**Beta**) reductions and the size of the initial term.

3.7 Example and Applications

Let $M \equiv \lambda y.((\lambda x.x)y)$. We then have

$$M \longrightarrow\!\!\!\!\rightarrow_\beta \lambda y.y$$

The equivalent term in **ACCL** after encoding variables, is given by

$$[\![M]\!]_{\mathrm{ACCL}} \equiv \Lambda(\mathrm{Apply}(\Lambda(0!),\ 0!)) \equiv \Lambda(\mathrm{Apply}(\Lambda(\mathrm{Var}),\ \mathrm{Var}))$$

We then have

$$\begin{aligned} &\Lambda(\mathrm{Apply}(\Lambda(\mathrm{Var}),\ \mathrm{Var})) \\ &\quad \longrightarrow\!\!\!\!\rightarrow_{(\mathbf{Beta})} \Lambda([\mathrm{Var},\ \langle \emptyset,\ \mathrm{Var}\rangle]) \\ &\quad \longrightarrow\!\!\!\!\rightarrow_{(\mathrm{VarRef})} \Lambda(\mathrm{Var}) \end{aligned}$$

and

$$[\![\Lambda(\mathrm{Var})]\!]_\lambda \equiv \lambda y.y$$

In essence, **ACCL** is just a formalization of the informal notions of closure and environment given in the introduction, coupled with a mechanism for indexing environments.

If we treat the axioms of **ACCL** as *transformation rules* on terms, we can note that opportunities for sharing of terms in practical reduction schemes are inherent in the rule. Meta-variables in the axioms may be treated as pointers to terms, and transformations on terms using the axioms as rules should simply copy the corresponding pointer when a meta-variable appears on both sides of the equation, rather

than copying the entire term. When a meta-variable is *repeated* on the right side of the equation, as with rules (**DApply**) and (**DE**), the term-pointers corresponding to the repeated variables may safely be set to point to the same term, creating graph-like structures. When any of the rules which contain a single meta-variable on the right side are applied, one has a choice of using *indirection* nodes of some sort or copying the topmost operator of the term.

I will not pursue a formal characterization of sharing here; an informal approach suffices for the purposes of the discussion here. More formal techniques for describing reduction using sharing have been proposed by Staples in [Sta80a,Sta80b,Sta80c,Sta81].

Figure 1 describes an algorithm, *rwhnf*(), that transforms a term of the form $[A, E]$ to the **ACCL** equivalent of weak head normal form, *WHNF*. It is very similar to the interpreters of Henderson and Morris [HM76]) and Aiello and Prini [AP81]. The algorithm is specified using rules of **ACCL**, a recursive redex selection strategy, and shared terms. Since this function simply applies **ACCL** rules to a term in a fixed order, Theorem 3.7 shows it to be correct (i.e., that it effectively performs β-reduction and nothing else). Though the algorithm is not fully-lazy in the sense of Wadsworth, it illustrates the simplicity with which interpreters can be specified using **ACCL**, and functions as a starting point for much more lazy interpreters that can be analyzed using $\mathbf{ACCL}^{\mathbf{L}}$.

The normalization properties of reduction schemes using **ACCL** depend on whether or not applications of the rule (**Beta**) are *needed*; this property is discussed below. *rwhnf*() does indeed turn out to be normalizing.

Given $M \in Ter(\lambda^{DB})$, we construct term $[[\![M]\!]_{ACCL}, \emptyset] \in Ter(\mathbf{ACCL})$, and reduce it to $(B \in WHNF) \equiv [[\![N]\!]_{ACCL}$. We thus have $M \twoheadrightarrow_\beta N, N \in whnf$. Figures 2, 3, and 4 are algorithms for normalizing environments (to "partial environment normal form," *PENF*).

The functional *notation* used in the algorithm should be reasonably self explanatory for someone familiar with a language such as ML or Miranda. However, the algorithm should be considered a recursively specified sequence of *transformations* on the term given as argument, not a true function, since no value is to be returned. The **case** statement executes various statements depending on a pattern to be matched. Subpatterns within larger patterns are named using the notation "*subpat*: *A*" Pattern variables represent pointers to terms, and if a pattern variable appears on the right side of a pattern, the pointer to the term represented by the variable, not the term itself, is copied. ":=" causes a term to be overwritten according to some rule of **ACCL**; only those parts of the overwriting term not named by pattern variables are newly allocated. Statements inside "**seq**...**endseq**" are executed in sequence. *copy*(A) copies the topmost operator of A; all of A's subterms are referred to by pointers in the new term.

```
rwhnf([L, E]: C) ≡
    case L of
        Apply(A, B): seq
            C := Apply([A, E]: A', [B, E]: B');
                                        {rule DApply}
            rwhnf(A');
            if A' ≡ [Λ(A''), E']
            then seq
                C := [A'', ⟨E', B'⟩];  {rule Beta'}
                rwhnf(C);
                endseq
            else skip;
        Λ(A): skip;                          {C ∈ WHNF}
        [L1, E1]: seq
            C := [L1, E1 ∘ E];               {rule AssC}
            rwhnf(C)
            endseq;
        (Var: L1): seq                            {C = 0!}
            rpenf(E);        {transform E to PENF}
            case E of
                ∅: C := L1;
                                             {rule NullC}
                □^n: skip;
                        {E = □^n, thus C ∈ WHNF}
                ⟨E, A⟩: seq
                    rwhnf(A);
                    C := copy(A);
                      {rule VarRef, C ∈ WHNF}
                    endseq
            endcase
            endseq
    endcase
endfn
```

Figure 1: Algorithm *rwhnf*()

```
rpenf(E) ≡
    case E of
        ∅: skip;                             {E ∈ PENF}
        □^n: skip;                           {E ∈ PENF}
        ⟨E1, A⟩: rpenf(E1);
        E1 ∘ E2: seq
            rpenf(E1);
            rpenf(E2);
            composeEnvs(E)
            endseq
    endcase
endfn
```

Figure 2: Algorithm *rpenf*()

```
composeEnvs((E1 ∘ E2): E) ≡                    {E1, E2 ∈ PENF}
    case E1 of
        ∅: E := E2;                                  {rule NullEL}
        □: distribShiftL(E);
        ((□: E3) ∘ E4): seq
            E := E3 ∘ ((E4 ∘ E2): E');               {rule AssE}
            composeEnvs(E');
            distribShiftL(E)
            endseq
        ⟨E3, A⟩: seq
            E := (⟨E3 ∘ E2⟩: E', [A, E2]);
                                                     {rule DE}
            composeEnvs(E')
            endseq
    endcase
endfn
```

Figure 3: Algorithm *composeEnvs*()

```
distribShiftL(((□: E1) ∘ E2): E) ≡              {E2 ∈ PENF}
    case E2 of
        ∅: E := E1;                                  {rule NullER}
        □^n: skip;                {E = □^{n+1} ∈ PENF, n > 0}
        ⟨E3, A⟩: E := E3                             {rule ShiftE}
    endcase
endfn
```

Figure 4: Algorithm *distribShiftL*()

4 Optimality Criteria

In [Lév78,Lév80], J.-J. Lévy studied the issue of optimal reduction in the λ-calculus in light of the previous work of Wadsworth on graph reduction. Lévy noted that by sharing redexes through graph structures, Wadsworth was essentially contracting multiple β-redexes in *parallel*. Lévy was able to define a natural class of parallel reductions on redexes that are essentially copies of one another, and specify criteria that would have to be satisfied by any optimal parallel reduction of sets such copies. The notion of copy Lévy had in mind was sets of identical terms, modulo substitutions for free variables. Such copies are exactly the terms created by the process of substituting the argument term for multiple instances of the binding variable in the body of a λ-term, and are formally known as *residuals*.

His critical observation was that by examining a term and the reduction that produced it (its "history"), it is decidable which sets of redexes in the term are copies of some redex, or more importantly, *could have been copies* in an alternate reduction (beginning and ending with the same term). He noted that by reducing maximal sets of such copies in parallel, an optimal reduction could be achieved. The question was then whether any practical reduction scheme could be implemented that would ensure that all such copies are shared, and thus for which contraction of a single term would effectively contract all copies. Lévy speculated that some scheme using shared closures, which permit contractions independent of substitutions for free variables (i.e., environments) might allow optimal reduction.

[Lév78] makes use of an extension to the λ-calculus that allows terms to be *labeled*. Such annotations allow specific terms to be "traced" as a reduction progresses, and provides means to compare different reductions. In addition, the labelings are modified during the course of a reduction in such a way that the reduction "history" of a particular term is evident on inspection. An alternative analysis in [Lév80] avoids labelings, and instead allows reductions to be compared using the idea of *meta-reduction*, or reduction *on* reductions to certain canonical forms. The analysis using labels provides a greater intuitive feel for the problem, and, more to the point, will simplify the proofs to follow. Therefore, I will review the analysis using labelings here.

4.1 Lévy's Labeled Lambda Calculus

Lévy's *labeled λ-calculus* was first introduced in [Lév75]. I will use a slightly simplified version proposed by Klop [Klo80], in which an extensive investigation of properties of reductions is made, much of which nicely complements the work of Lévy. A concise summary of Lévy's labeled λ-calculus is given in [Bar84, p. 382, Ex. 14.5.5], and a summary of a number of useful properties is given in [BKKS87, Appendix].

First we must define what constitutes a *label*:

Definition 4.1 *The set of* Lévy-labels, *designated* L, *is defined inductively as follows:*

$$l \in S \implies l \in \mathrm{L}$$

$$w, v \in \mathrm{L} \implies wv \in \mathrm{L}$$
$$w \in \mathrm{L} \implies \underline{w} \in \mathrm{L}$$

where $S = \{a, b, c, \ldots\}$ is an infinite set of symbols) and wv is the concatenation of labels w and v.

An *atomic* label is a label consisting of a single symbol. Note that nested underlinings, e.g. $\underline{\underline{abcd}}$, may occur.

The set of *labeled* λ-terms consists of the regular λ-terms and terms annotated with labels:

Definition 4.2 *The set of terms in Lévy's* labeled λ-calculus, *designated $Ter(\lambda^{\mathrm{L}})$, is defined as follows:*

$$M \in Ter(\lambda) \implies M \in Ter(\lambda^{\mathrm{L}})$$
$$M \in Ter(\lambda^{\mathrm{L}}), w \in \mathrm{L} \implies (M^w) \in Ter(\lambda^{\mathrm{L}})$$

where x is an arbitrary variable.

If M is a meta-variable referring to a labeled term, M^w denotes the concatenation of w to the label of the term to which M refers. I will often refer to terms "with" or "having" label w. A term M *has* label w if M is of the form N^w and N is not of the form P^u for non-null label u. The parentheses surrounding a labeled term will often be omitted for the sake of clarity if no confusion would arise. (If, however, a parenthesized term is *itself* labeled, a formal reduction rule is required to eliminate the parentheses; see below.)

In contexts where a labeled term is expected, unlabeled terms will be treated as having the *null label*, ϵ. We define label concatenation and underlining to behave on the null label as follows:

$$\epsilon w = w$$
$$w\epsilon = w$$
$$\underline{\epsilon} = \epsilon$$

The label of the abstraction part of a redex is called the *degree* of the redex. Thus the degree of the (Ix) redex in $((\lambda x.(I^a x^b)^c)^d z^e)^f$ is a (not c).

The *β-contraction* rule is now defined for labeled terms as follows:

Definition 4.3 *Labeled β-contraction, denoted by $\longrightarrow_{\beta\mathrm{L}}$, is a relation on members of $Ter(\lambda^{\mathrm{L}})$defined by:*

$$C[\,((\lambda x.M)^w N)^v\,] \longrightarrow_{\beta\mathrm{L}} C[\,(M^{\underline{w}}[x := N^{\underline{w}}])^v\,]$$

where C is an arbitrary context and M and N are arbitrary members of $Ter(\lambda^{\mathrm{L}})$.

Note that with the null label convention, labeled β-contraction is exactly the same as regular β-contraction on unlabeled terms.

Though the labeled β-contraction rule looks a bit formidable, the idea is quite simple: Whenever a redex is contracted, the underlined form of the label of the redex's abstraction (w) is attached both to the body of the abstraction (M) and to all instances of the argument (N) substituted into the body. Any label attached to the application term (v) is left intact. The attachment to a label of an underlined substring, say ($\underline{w}$), is an indication that the term was effectively generated by contraction of a redex having degree w (this assumes, as I always will, that any labeled reduction has an initial term with no underlined labels). One can thus view labels as a sort of genetic code, in the sense that by knowing the labels of the initial term ("matriarch"?) of a reduction, the lineage of a subsequent term in the reduction may be traced by inspection of the labels.

The formation rules of $Ter(\lambda^{\mathrm{L}})$ allow multiple labelings of parenthesized terms, which can be created as a resulted of labeled β-contraction. This requires an auxiliary reduction rule for labels:

Definition 4.4 *The* label simplification rule, $\longrightarrow_{\mathrm{lab}}$, *is the following relation on members of $Ter(\lambda^{\mathrm{L}})$:*

$$C[\,(M^w)^v\,] \longrightarrow_{\mathrm{lab}} C[\,M^{wv}\,]$$

where C is an arbitrary context and M^w is a term of $Ter(\lambda^{\mathrm{L}})$.

We then have:

Definition 4.5 Labeled β-reduction, $\twoheadrightarrow_{\beta\mathrm{L}}$, *is the reflexive, transitive closure of* $(\longrightarrow_{\mathrm{lab}} \cup \longrightarrow_{\beta\mathrm{L}})$, *where '$\cup$' denotes relational union.*

The label simplification rule is a technical necessity, but a practical nuisance. Without loss of generality, when referring to a labeled term, I will assume it has been simplified as much as possible using $\twoheadrightarrow_{\beta\mathrm{L}}$. This assumption is technically justified by the following theorem:

Theorem 4.1 ([Lév75]) $\twoheadrightarrow_{\beta\mathrm{L}}$ *has the Church-Rosser (confluence) property, i.e.,*

$$M \twoheadrightarrow_{\beta\mathrm{L}} N_1 \wedge M \twoheadrightarrow_{\beta\mathrm{L}} N_2 \implies (\exists P) N_1 \twoheadrightarrow_{\beta\mathrm{L}} P \wedge N_2 \twoheadrightarrow_{\beta\mathrm{L}} P$$

Thus labeled λ-reduction is as "well-behaved" as its unlabeled counterpart, and, in a sense, is a strict refinement of the regular λ-reduction. Ignoring the labels, it is simply regular λ-reduction. Depending on the initial labeling, however, it can give a great deal more information about the reduction process.

We can now define transformations from the unlabeled to the labeled world and vice versa:

Definition 4.6 *Let M^l be a term of $Ter(\lambda^{\mathrm{L}})$. Then the* erasure *of M^l, $Er(M^l)$ is the same term with all the labels erased.*

Definition 4.7 *Let M be a term of $Ter(\lambda)$. Then $M^l \in Ter(\lambda^{\mathrm{L}})$ is a* labeling *of M iff $Er(M^l) = M$.*

We can also define the erasure of a reduction (overloading the meaning of '$Er()$'):

Definition 4.8 *Let σ^l be a labeled reduction. Then the* erasure *of σ^l, $Er(\sigma^l)$, is the unlabeled reduction obtained by erasing the labels of all the terms in the reduction and replacing all labeled β-contractions by unlabeled β-contractions.*

Finally, we can "lift" reductions on unlabeled terms to their labeled counterparts:

Definition 4.9 *Let M be a term of $Ter(\lambda)$, M^l be some labeling of M, and $\sigma: M \longrightarrow\!\!\!\!\rightarrow_\beta N$. Then the lifted reduction $Lift(\sigma, M^l)$ is defined as the labeled reduction with initial term M^l in which the redexes contracted are the labeled counterparts of those contracted in σ.*

4.2 Optimality

4.2.1 Labels and Residuals

With the machinery of the labeled λ-calculus at hand, certain definitions that are rather complicated without it become straightforward. Labelings can be used to divide all the redexes in a reduction into equivalence classes based on their label. Such equivalence classes are deemed redex *families*:

Definition 4.10 ([Lév78]) *Let*

$$\rho: M \longrightarrow\!\!\!\!\rightarrow_\beta N$$

be a reduction. Let l be a labeling of M such that each subterm of M^l has a unique *atomic label. Let*

$$\rho^l: M^l \longrightarrow\!\!\!\!\rightarrow_{\beta L} N^l = Lift(\rho, M^l)$$

be the labeled version of ρ. Then a redex R_j in any term of ρ (not necessarily a redex contracted by ρ) is a member of family class F^l_w *iff the corresponding redex R^l_j in ρ^l has degree w.*

Rather remarkably, it turns out that family classes can consist not only of sets of redexes that are effectively copies (i.e., residuals) of terms in the current reduction, but also may consist of sets of redexes that are not residuals of any redex in the current reduction, but *would* be residuals in a *different* reduction with the same initial and final terms. Thus labeling makes evident on inspection a property that might seem to require enumeration of all reductions.

4.2.2 Redex Sharing and Parallel Reductions

Having demonstrated the usefulness of the labeled λ-calculus, we can now formalize the notion of *sharing* of terms. Lévy noted that the reduction of a shared redex could be viewed as a *parallel* reduction of all the redexes represented by the shared term in its "flattened," non-graphical form. For instance, in Example 2.2 above, the shared contraction of the (Iz) redex may be viewed as the parallel contraction of the two terms that share it:

Example 4.1

$$\sigma_1'': (\lambda y.(yy))(Iz) \longrightarrow_\beta (Iz)(Iz) \longrightarrow\!\!\!\!\rightarrow_{\|\beta} zz$$

where '$\longrightarrow\!\!\!\!\rightarrow_{\|\beta}$' represents parallel β-contraction. Note that parallel β-contraction subsumes ordinary 0 or 1 step β-reduction ($\longrightarrow\!\!\!\!\rightarrow_{\equiv\beta}$), which is a development of 0 or 1 redexes.

Parallel *reductions* are represented thus:

$$\sigma: M_0 \xrightarrow{C_1}\!\!\!\!\rightarrow_{\|\beta} M_1 \xrightarrow{C_2}\!\!\!\!\rightarrow_{\|\beta} \cdots \xrightarrow{C_n}\!\!\!\!\rightarrow_{\|\beta} M_n$$

where the C_i are the sets of redexes in M_i contracted in parallel at each step.

Defining a consistent notion of parallelism for *overlapping* redexes requires a bit of care. Formally, Lévy defines a parallel reduction as the *complete development* of a set of redexes. See [Lév78] or [Lév80] for more details.

We can now define parallel reductions that reduce entire family classes at once:

Definition 4.11 (Lévy) *A parallel reduction*

$$\sigma: M_0 \xrightarrow{F_{w_1}}\!\!\!\!\rightarrow_{\|\beta} M_1 \xrightarrow{F_{w_2}}\!\!\!\!\rightarrow_{\|\beta} \cdots \xrightarrow{F_{w_n}}\!\!\!\!\rightarrow_{\|\beta} M_n$$

is family-complete *iff for each M_i, F_{w_i} is the set of all members of some redex family F_w in M_i.*

4.2.3 Call-By-Need Reductions

In order to ensure that an optimal reduction does no unnecessary work (although perhaps does it quite efficiently), we need to ensure that any optimal reduction, like leftmost reduction, reduces no *unneeded* redex. This leads to the following formal definitions:

Definition 4.12 (Lévy) *A redex R in some expression $M \in Ter(\lambda)$ is* needed *iff, for all terminating reductions σ with initial term M, either R or one of its residuals is contracted in σ.*

Definition 4.13 (Lévy) *A parallel reduction*

$$\rho: M_0 \xrightarrow{C_1}\!\!\!\!\rightarrow_{\|\beta} M_1 \xrightarrow{C_2}\!\!\!\!\rightarrow_{\|\beta} \cdots \xrightarrow{C_n}\!\!\!\!\rightarrow_{\|\beta} M_n$$

is a call-by-need *reduction iff there is at least one needed redex in each C_i.*

We now have Lévy's optimality theorem:

Theorem 4.2 ([Lév78,Lév80]) *A parallel reduction*

$$\rho: M \longrightarrow\!\!\!\!\rightarrow_{\|\beta} N$$

is optimal *for the class of all parallel reductions with initial term M if*

- *ρ is* family-complete.
- *ρ is* call-by-need.

Note that the theorem does not *require* that an optimal strategy use shared redexes—a regular (non-parallel) β-contraction is a degenerate parallel contraction, and if all family classes have one member, a complete reduction requires contraction of only one redex at a time. However, if a fixed redex selection strategy is to be used, some form of sharing is inevitable.

5 Labeled ACCL

By analogy with Lévy's labeled λ-calculus, we can define a similarly labeled version of **ACCL**, $\mathbf{ACCL}^{\mathrm{L}}$.

Definition 5.1 *The axioms of* $\mathbf{ACCL}^{\mathrm{L}}$ *are as follows:*

(Beta)	$\mathrm{Apply}(\Lambda(A)^w, B)^u = [A^{u\underline{w}}, \langle \emptyset, B^{\underline{w}} \rangle]$
(AssC)	$[[A, E_1]^u, E_2]^v = [A^{uv}, E_1 \circ E_2]$
(NullEL)	$\emptyset \circ E = E$
(NullER)	$E \circ \emptyset = E$
(ShiftE)	$\Box \circ \langle E, A \rangle = E$
(VarRef)	$[\mathrm{Var}^u, \langle E, A \rangle]^v = A^{uv}$
(DA)	$[\Lambda(A)^u, E]^v = \Lambda([A, \langle E \circ \Box, \mathrm{Var} \rangle])^{uv}$
(DE)	$\langle E_1, A \rangle \circ E_2 = \langle E_1 \circ E_2, [A, E_2] \rangle$
(DApply)	$[\mathrm{Apply}(A, B)^u, E]^v = \mathrm{Apply}([A, E], [B, E])^{uv}$
(AssE)	$(E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3)$
(NullC)	$[A^u, \emptyset]^v = A^{uv}$
(DLabel)	$[A, E]^u = [A^u, E]$

Note that the **DLabel** has no analogue in unlabeled **ACCL**. It is the $\mathbf{ACCL}^{\mathrm{L}}$ equivalent of the the convention allowing the removal of parentheses in multiply-labeled parenthesized terms of λ^{L}. By analogy with the labeled λ-calculus, the *degree* of a labeled **(Beta)** redex is the label of its abstraction term; e.g., $\mathrm{Apply}(\Lambda(A^u)^w, B^v)^z$ has degree w.

Theorems 3.4, 3.5, 3.9, and 3.7 all apply to $\mathbf{ACCL}^{\mathrm{L}}$ and λ^{L}; the proofs are quite similar and are omitted. The translations $[\![\cdot]\!]_{\lambda^{\mathrm{L}}}$ and $[\![\cdot]\!]_{\mathbf{ACCL}^{\mathrm{L}}}$ are defined in the obvious way analogous to their unlabeled counterparts.

We can now apply Lévy's optimality criteria directly to reductions in **ACCL**, using $\mathbf{ACCL}^{\mathrm{L}}$. The idea is to consider each **(Beta)** contraction in a term A as representing a parallel β-contraction on the corresponding λ-term $[\![Inf(A)]\!]_\lambda$.

We can then make the following definitions:

Definition 5.2 *A reduction* $A \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ACCL}^{\mathrm{L}}} B$ *is* λ-optimal *if the number of* **(Beta)** *contractions therein is less than or equal the number of parallel* β*-contractions in an optimal* λ*-reduction from* $[\![Inf(A)]\!]_\lambda$ *to* $[\![Inf(B)]\!]_\lambda$.

Definition 5.3 *A* **(Beta)** *redex* B *in a term* A *is* λ-needed *iff the* λ*-equivalent of one of* B*'s residuals* $B_i' \in Resid(B, A)$, $[\![B_i']\!]_\lambda$, *is needed in* $[\![Inf(A)]\!]_\lambda$ *in the sense of Definition 4.12.*

Theorem 5.1 *Let* $A^l \in \mathbf{ACCL}^{\mathrm{L}}$ *be a term all of whose subterms have unique labels. Let* $\rho^l : A^l \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ACCL}^{\mathrm{L}}} B^l$ *be a* $\mathbf{ACCL}^{\mathrm{L}}$ *reduction. Then the corresponding unlabeled reduction* $\rho : Inf(A) \longrightarrow\!\!\!\!\!\rightarrow_{\mathbf{ACCL}} Inf(B)$ *is* λ*-optimal if no two* **(Beta)** *redexes in* ρ^l *have the same degree and each* **(Beta)** *redex is* λ*-needed.*

Proof Follows from Lévy's results on the labeled λ-calculus, the labeled form of Lemma 3.9 and Theorem 3.8. □

The above theorem gives us the promised tool for analysis of laziness. If we construct a λ-interpreter whose action can be expressed in terms of some application of the rules of **ACCL**, we can determine how close to optimality any such interpreter can come by showing how many **(Beta)** redexes in the corresponding labeled reductions have the same label.

6 Non-Optimality of Reduction with Shared Closures

I can now show that there is *no* λ-optimal reduction possible in $\mathbf{ACCL}^{\mathrm{L}}$, and thus that no reduction scheme that can be expressed using the axioms of **ACCL** is optimal in Lévy's sense. I do so by exhibiting a λ-term for which *every* **ACCL** reduction causes more than one **(Beta)** redex to be contracted in the corresponding labeled form, even when when all terms are shared that are permissible under **ACCL**'s rules. The term is as follows:

$$(\lambda x.((x A^d)(x B^e)))(\lambda y.((\lambda z.(z^a t)(z^b u))(\lambda w.y^c v)))$$

where A and B are arbitrary λ-abstractions. Not all subterms are given labels for the sake of clarity.

Space does not permit a complete enumeration of all possible reductions of its corresponding $\mathbf{ACCL}^{\mathrm{L}}$ translation. However, the crux of the matter is embodied in the following term, which must be produced in any reduction of the **ACCL** equivalent of the term above if no prior **(Beta)** redexes with the same label are to be reduced twice:

$$(([\bullet, \langle\!\langle y := A^d \rangle\!\rangle] : C_1)([\bullet, \langle\!\langle y := B^e \rangle\!\rangle] : C_2))$$
$$\qquad [((z^a t)(z^b u)),\ \langle\!\langle z := \lambda w.(y^c v) \rangle\!\rangle] : C_3$$

As before, $\langle\!\langle \cdot \rangle\!\rangle$ represents a **ACCL** environment with the bound variable indicated explicitly. The notation $T : N$ is used to give names to subterms. One is forced here to choose between reducing closure C_3 or one of closures C_1 or C_2. Choosing C_3 yields:

$$(([\bullet, \langle\!\langle y := A^d \rangle\!\rangle] : C_1)([\bullet, \langle\!\langle y := B^e \rangle\!\rangle] : C_2))$$
$$\qquad ((y^c v)(y^c v))$$

which reduces to

$$(((A^{dc} v)(A^{dc} v))([\bullet, \langle\!\langle y := B^e \rangle\!\rangle] : C_2))$$
$$\qquad ((y^c v)(y^c v))$$

in which two redexes of the form $(A^{dc} v)$ are created, thus yielding a non-optimal reduction (since they have the same degree and are no longer shared).

To avoid the copying that occurs above, one could alternately first reduce closure C_1 (or C_2, for which the argument to follow is symmetric), which would eventually yield a term of the following following form:

$$(([\bullet, \langle\!\langle z := \lambda w.(A^{dc} v) \rangle\!\rangle] : C_1')([\bullet, \langle\!\langle z := \lambda w.(B^{ec} v) \rangle\!\rangle] : C_2'))$$
$$\qquad ((z^a t)(z^b u))$$

which reduces to

$$((\lambda w.\bullet)^a t)((\lambda w.\bullet)^b t)((\lambda w.\bullet)^a t)((\lambda w.\bullet)^b t)$$
$$\qquad (A^{dc} v) \qquad\qquad (B^{ec} v)$$

The term above has two (actually, two sets) of unshared redexes with the same degree, e.g., $((\lambda w.(A^{dc} v))^a t)$ and

$((\lambda w.(B^{cc}v))^a t)$. If both are needed (which depends on the particular abstractions chosen for A and B, a non-optimal reduction will once again result. In the end, no matter what choice is made, a non-optimal reduction occurs.

The informal observation that shared closures and environments alone are insufficient to implement optimal reduction schemes was also made independently by Curien in [Cur86c]. He did not, however, provide a formal connection (such as that made above using labels) between redex families in the lambda calculus and their equivalents in a formal system using environments, nor was the system he was using as general as the one proposed here.

7 Related Work and Conclusions

A system almost identical to **ACCL** has been independently proposed by Abadi, et.al. [ACCJL89]. Its term structure is isomorphic to that of **ACCL**, and its axioms are the same with two minor exceptions. They propose to use their system to study properties of substitutions, to describe type-checking algorithms, and as the basis for machine-oriented implementations of reduction schemes. They have not, however, proposed a labeled system for the study of the optimality problem.

[AKP84] provides an analysis of the differences between various lazy and fully-lazy λ-interpreters without examining the issue of optimality.

Two schemes, by Staples [Sta82] and, recently, by Lamping [Lam89], have been proposed that claim to implement optimal λ-reduction. Both seem to allow terms to be shared that traditional environment or substitution mechanisms do not allow. However, they are notable for their extreme complexity, and it is not clear that the overhead incurred by these schemes in order to ensure family classes are always shared is not prohibitive.

A practical optimal reduction mechanism might indeed exist for a restricted class of λ-terms, e.g., the so-called "supercombinators" used in functional programming. However, if one believes that **ACCL** is a sufficiently general model of reduction using shared environments or closures, then one must conclude that shared environments, closures, or terms alone are insufficient to achieve optimality in a practical interpreter.

To summarize, I have described a new system of combinators, **ACCL**, with which one can describe a wide variety of reduction methods for the λ-calculus using sharing. I have proved that essentially any reduction in **ACCL** corresponds to β-reduction in the λ-calculus, and thus that λ-reduction schemes using **ACCL** may be proved correct trivially. I have also described a labeled variant of **ACCL**, $\mathbf{ACCL}^{\mathrm{L}}$, which can be used as a tool to analyze the degree of laziness present in reduction schemes. I have shown, however, that **ACCL** is insufficient for implementing *optimal* reduction schemes, and thus that more than shared closures, environments, or λ-terms are apparently necessary if optimality is to be achieved at all.

8 Acknowledgements

I would like to thank Tim Teitelbaum for his support, encouragement, and productive discussions during the genesis of these ideas. I am also grateful to Pierre-Louis Curien for his comments on an earlier version of this paper and to Martin Abadi for supplying me with an unpublished version of his joint paper. Finally, I would like to thank especially Jean-Jacques Lévy and Thérèse Hardin for fruitful conversation, providing helpful comments, and pointing me toward related work.

A The Lambda Calculus and Term Rewriting Systems

I will briefly review some of the notation for the lambda-calculus used herein. The conventions used here will generally follow those of [Bar84], to which the reader is referred for details, although a few are taken from [Klo80] or [BKKS87].

A.1 Notation

$C[M]$ denotes a *context* containing M, i.e., C[M] is a λ-term with designated subterm M. M need not be a proper subterm of $C[M]$. Contexts may be defined similarly for other rewriting systems.

$M[x := N]$ denotes the result of substituting N for all free occurrences of x in M.

β-contraction is denoted by $\longrightarrow_\beta$.

The reflexive, transitive closure of $\longrightarrow_\beta$, *β-reduction*, is denoted by $\longrightarrow\!\!\!\!\rightarrow_\beta$.

Other notions of reduction will be defined using. analogous notation: if $\longrightarrow_R$ is a relation, then $\longrightarrow\!\!\!\!\rightarrow_R$ will denote its reflexive, transitive closure, and $=_R$ the induced equivalence.

Since '=' will be reserved to represent equality induced by a reduction relation, I will use '$\equiv$' to denote syntactic identity of λ-terms. I will identify on the syntactic level terms that are identical modulo changes of bound variable and avoid the machinery of *α-conversion*, i.e., I will feel free to say

$$\lambda x.x \equiv \lambda y.y$$

(As a practical matter, some reduction schemes will require a mechanism that effectively performs renaming. Such a mechanism will be introduced later).

Reductions, sequences of β-contractions, will be denoted as follows:

$$\sigma\colon M_0 \xrightarrow{R_1} M_1 \xrightarrow{R_2} M_2 \xrightarrow{R_3} \cdots \xrightarrow{R_n} M_n$$

σ designates the the entire reduction sequence. The M_i are the *terms* of the reduction. The R_i denote the redexes contracted at each step. Where clear from context, the R_i may be omitted. Occasionally, it will be convenient to elide the intermediate terms and denote the entire sequence by $\sigma\colon M_0 \longrightarrow\!\!\!\!\rightarrow_\beta M_n$.

A.2 The de Bruijn lambda calculus

The de Bruijn λ-calculus [dB72,dB78] is a variant of the λ-calculus in which variables are replaced by *de Bruijn numbers* denoting their binding depth in the term in which they are contained. This facilitates reduction without concern for variable "capture," which can occur during conventional λ-reduction even when the initial term of a reduction contains no bound variables with the same name. By providing a variable substitution mechanism that appropriately adjusts the de Bruijn numbers of substituted terms, the de Bruijn λ-calculus eliminates the need for α-conversion. The following definitions are from [Cur86a]

Definition A.1 *The set of terms in the* de Bruijn λ-calculus, *designated* $Ter(\lambda^{DB})$, *is defined inductively as follows*

$$\begin{aligned} n \in \mathcal{N} &\implies n \in Ter(\lambda^{DB}) \\ M, N \in Ter(\lambda^{DB}) &\implies (MN) \in Ter(\lambda^{DB}) \\ M \in Ter(\lambda^{DB}) &\implies \lambda.M \in Ter(\lambda^{DB}) \end{aligned}$$

where $\mathcal{N}$ *is the set of natural numbers.*

Definition A.2 *For any* $M \in Ter(\lambda)$ *such that* $FV(M) \subseteq \{x_0, \ldots, x_n\}$, *define its de Bruijn translation,* $M_{DB(x_0,\ldots,x_n)} \in Ter(\lambda^{DB})$, *as follows:*

$$\begin{aligned} x_{DB(x_0,\ldots,x_n)} &= i, \text{ where } i \text{ is minimum s.t. } x = x_i \\ (\lambda y.M)_{DB(x_0,\ldots,x_n)} &= \lambda.M_{DB(y,x_0,\ldots,x_n)} \\ (MN)_{DB(x_0,\ldots,x_n)} &= M_{DB(x_0,\ldots,x_n)} N_{DB(x_0,\ldots,x_n)} \end{aligned}$$

(I will usually write M_{DB} rather than $M_{DB(x_0,\ldots,x_n)}$ when the free variable ordering is irrelevant).

Substitution, β-reduction, and η-reduction can be suitably redefined on λ^{DB} such that

$$M \longrightarrow_{\beta\eta} N \iff M_{DB} \longrightarrow_{\beta\eta} N_{DB}$$

For a concise exposition of the details of β-reduction and the substitution process, see [Cur86a]

References

[ACCJL89] M. Abadi, L. Cardelli, P.-L. Curien, and J.J.-Levy. Explicit substitutions. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, 1989.

[AKP84] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of computation in functional language implementations. In *Proc. International Workshop on High-Level Computer Architecture*, Los Angeles, 1984.

[AP81] Luigia Aiello and Gianfranco Prini. An efficient interpreter for the lambda calculus. *Journal of Computer and System Sciences*, 23:383–424, 1981.

[Aug84] L. Augustsson. A compiler for Lazy ML. In *Proc. ACM Symp. on Lisp and Functional Programming*, Austin, 1984.

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[BBKV76] H.P. Barendregt, J. Bergstra, J.W. Klop, and H. Volken. Degrees, reductions, and representability in the lambda calculus. Preprint 22, Department of Mathematics, University of Utrecht, The Netherlands, 1976.

[BKKS87] H.P. Barendregt, J.R. Kennaway, J.W. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75:191–231, 1987.

[CCM87] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ, 1941.

[Cur86a] P.-L. Curien. Categorical combinators. *Information and Control*, 69:188–254, 1986.

[Cur86b] P.-L. Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman, London, 1986.

[Cur86c] P.-L. Curien. De la difficulté d'implémenter le partage optimal au sens de lévy. Unpublished Note, Université de Paris VII, 1986.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Proc. of the Koninklijke Nederlandse Akademie van Wetenschappen*, 75(5):381–392, 1972.

[dB78] N.G. de Bruijn. Lambda calculus with name-free formulas involving symbols that represent reference transforming mappings. *Proc. of the Koninklijke Nederlandse Akademie van Wetenschappen*, 81(3):348–356, 1978.

[Der87] Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, England, 1988.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 34–45, Portland, 1987. Springer-Verlag. Lecture Notes in Computer Science 274.

[Har87] Thérèse Hardin. *Résultats de Confluence pour les Règles Fortes de la Logique Combinatoire Catégorique et Liens avec les Lambda-Calculs*. PhD thesis, Université de Paris VII, 1987.

[Har89] Thérèse Hardin. Confluence results for the pure strong categorical logic CCL. λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[HL86] Thérèse Hardin and Alain Laville. Proof of termination of the rewriting system SUBST on CCL. Rapports de Recherche 560, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France, August 1986.

[HM76] P. Henderson and J.H. Morris. A lazy evaluator. In *Proc. Third ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[HO80] G. Huet and D.C. Oppen. Equations and rewrite rules: A survey. In R.V. Book, editor, *Formal Language Theory, Perspectives, and Open Problems*, pages 349–405. Academic Press, London, 1980.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 1986.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Hug84] R.J.M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, Oxford University, September 1984. (PRG-40).

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proc. ACM Conf. on Compiler Construction*, Montreal, 1984.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proc. Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985. Lecture Notes in Computer Science 201.

[KL80] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path orderings. Unpublished note, Department of Computer Science, Univerity of Illinois, Urbana, IL, 1980.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematical Centre, Kruislaan 413, Amsterdam 1098SJ, The Netherlands, 1980.

[Lam89] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, 1989.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[Les84] Pierre Lescanne. Uniform termination of term rewriting systems. In B. Courcelle, editor, *Ninth Colloquium on Trees in Algebra and Programming*, pages 181–191, Bordeaux, France, 1984. Cambridge University Press.

[Lév75] Jean-Jacques Lévy. An algebraic interpretation of the λβK-calculus and a labelled λ-calculus. In C. Böhm, editor, *Proc. Symp. on λ-Calculus and Computer Science Theory*. Springer-Verlag, 1975. Lecture Notes in Computer Science 37.

[Lév78] Jean-Jacques Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris VII, 1978. (Thèse d'Etat).

[Lév80] Jean-Jacques Lévy. Optimal reductions in the lambda-calculus. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, London, 1980.

[Lin87] R.D. Lins. On the efficiency of categorical combinators as a rewriting system. *Software—Practice and Experience*, 17(8):547–559, August 1987.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, Englewood Cliffs, New Jersey, 1987.

[Rus87] Michael Rusinowitch. Path of subterms ordering and recursive decomposition ordering revisited. *J. Symbolic Computation*, 3:117–131, 1987.

[Sta80a] John Staples. Computation on graph-like expressions. *Theoretical Computer Science*, 10:171–185, 1980.

[Sta80b] John Staples. Optimal evaluations of graph-like expressions. *Theoretical Computer Science*, 10:297–316, 1980.

[Sta80c] John Staples. Speeding up subtree replacement systems. *Theoretical Computer Science*, 11:39–47, 1980.

[Sta81] John Staples. Efficient combinatory reduction. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 27:391–402, 1981.

[Sta82] John Staples. Two-level expression representation for faster evaluation. In *Proc. Second International Workshop on Graph Grammars and Their Applications*, pages 392–404. Springer-Verlag, 1982. Lecture Notes in Computer Science 153.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Wad71] C.P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.

[Yok89] Hirofumi Yokouchi. Church-rosser theorem for a rewriting system on categorical combinators. *Theoretical Computer Science*, 65:271–290, 1989.