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

John Field

TR 90-1091
May 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This is a revised version of a paper presented at the Seventeenth ACM Symposium
on Principles of Programming Languages, San Francisco, January 1990. This
research was supported by NSF and ONR under NSF Grant CCR-8514862, and by
ONR under contract number N00014-88-K-0594.
On Laziness and Optimality in Lambda Interpreters: 
Tools for Specification and Analysis*

John Field
Cornell University

Abstract

In this paper, we introduce a new formal system, ACCL, based on Curien's Categorical Combinators [Cur86a]. We show that ACCL has properties that make it especially suitable for analysis and implementation of a wide range of \(\lambda\)-reduction schemes using shared environments, closures, or \(\lambda\)-terms. In particular, the term structure of ACCL is very closely related to the structure of existing abstract machines for \(\lambda\)-reduction. ACCL is powerful enough to mimic arbitrary (strong) reduction in the \(\lambda\)-calculus, yet in contrast to the systems in [Cur86a] it is also confluent (on ground terms). As an example of the practical utility of this formalism, we use it to specify a simple lazy interpreter for the \(\lambda\)-calculus, whose correctness follows trivially from the properties of ACCL.

We then describe a labeled variant of ACCL, ACCL\(^L\), which can be used as a tool to determine the degree of "laziness" possessed by various \(\lambda\)-reduction schemes. In particular, ACCL\(^L\) is applied to the problem of optimal reduction in the \(\lambda\)-calculus. A reduction scheme for the \(\lambda\)-calculus is optimal if the number of redex contractions that must be performed in the course of reducing any \(\lambda\)-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 \(\lambda\)-terms. However, using ACCL\(^L\), we show that the sharing allowed by environments and closures in ACCL as implemented using standard term graph-rewriting techniques [BvEG+87] is insufficient to implement optimal reduction.

*This is a revised version of a paper presented at the Seventeenth ACM Symposium on Principles of Programming Languages, San Francisco, January, 1990. This research was supported by NSF and ONR under NSF grant CCR8514862, and by ONR under contract number N00014–88–K–0594.
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 $\lambda$-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 $\lambda$-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 $\beta$-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 $\beta$-reduction without sharing) to optimal, with full-laziness actually somewhere in between. Unfortunately, the exact nature of laziness in various implementations has apparently heretofore been something of a mystery\(^2\).

The aim of this paper is twofold: first, to provide a formalism that accurately reflects the reduction mechanisms in common use for implementation of languages based on the $\lambda$-calculus; second, to characterize precisely the behavior of such implementations with regard to the issues of laziness and optimality and to investigate their limitations.

This paper presupposes a familiarity with the $\lambda$-calculus [Chu41,Bar84,HS86], the de Bruijn $\lambda$-calculus [dB72,dB78,Cur86a], and basic ideas from term-rewriting systems [HO80,Hue80,Der87,Klo80]. A brief review of some of the terminology for these subjects is provided in Appendix B. An acquaintance with Curien’s Categorical Combinators [Cur86a,Cur86b,CCM87], and with the work of Lévy on optimality [Lév78,Lév80] would also be useful.

\(^1\)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 $\lambda$-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

\[
\begin{align*}
\sigma_1 : M & \rightarrow (Iz)(Iz) \rightarrow z(Iz) \rightarrow zz \\
\sigma_2 : M & \rightarrow (Iz)(Iz) \rightarrow (Iz)z \rightarrow zz \\
\sigma_3 : M & \rightarrow (\lambda y.(yy))z \rightarrow zz
\end{align*}
\]

$\sigma_1$ is a leftmost reduction—one where the leftmost redex is contracted at each step. $\sigma_3$ 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 $\sigma_3$ 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 $\sigma_1$ and $\sigma_2$, 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

\[
\begin{align*}
\sigma'_1 : (\lambda y.(yy))(Iz) & \rightarrow (\bullet \bullet) \rightarrow (\bullet \bullet) \equiv zz \\
& \quad \downarrow (Iz) \quad \downarrow z
\end{align*}
\]

$\sigma'_1$ proceeds from left to right, analogous to $\sigma_1$. 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 $N \equiv (N_1N_2)$, 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

\[ \rho_1: N \rightarrow (\bullet w)(\bullet z) \rightarrow (Iw)(\bullet z) \rightarrow w(\bullet z) \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \rightarrow w(Iz) \rightarrow wz \]

\[ \rho_2: N \rightarrow (\bullet w)(\bullet z) \rightarrow (\bullet w)(\bullet z) \rightarrow w(\bullet z) \rightarrow wz \]

Wadsworth's algorithm performs reduction \( \rho_1 \), while \( \rho_2 \) 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 \( \rho_2 \), 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 [BKK87]).

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 \( \rho_1 \) 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 \( \lambda \)-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 \( \lambda \)-abstraction after \( \beta \)-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 $\beta$-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:

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

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

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

where $B_1, B_2$, etc. are bindings.

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

**Example 2.4**

$$(\lambda y.(yy))(Iz) \rightarrow [(yy), \langle y := (Iz) \rangle]$$

$$\rightarrow ([y, \bullet][y, \bullet])$$

$$\langle y := (Iz) \rangle$$

$$\rightarrow ([y, \bullet][y, \bullet]) \cdots \rightarrow (zz)$$

$$\langle y := z \rangle$$

Use of closures obviates copying any part of the body of an abstraction after $\beta$-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 $N \equiv (N_1N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then, using shared environments, we have:
Example 2.5

\[
N \rightarrow [(xw)(xz), \langle x := N_2 \rangle] \\
\rightarrow ((\langle x := \varnothing \rangle [w, \varnothing]) [(xz), \varnothing]) \\
\rightarrow (\langle x := \varnothing \rangle \lambda y. (Iy)) \\
\rightarrow (\langle x := \varnothing \rangle \lambda y. (Iy)) (Iy) \\
\rightarrow (\langle x := \varnothing \rangle \lambda y. (Iy)) (Iy) \\
\rightarrow \ldots
\]

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 \(\lambda\)-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 \(\lambda\)-calculus, and thus has aspects in common with the de Bruijn \(\lambda\)-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 confluent system strong enough to simulate arbitrary \(\beta\)-reductions in the \(\lambda\)-calculus that could itself be simulated using only \(\beta\)-reduction. If such a system were available, it would provide an immediate proof of correctness for any reduction
scheme for the \( \lambda \)-calculus based on it (since any combinator reduction would correspond to a \( \beta \)-reduction). \( \lambda \)-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 and analyzing \( \lambda \)-reduction using environments, we will define a new 2-sorted equational theory, ACCL, akin to Curien’s theory CCL\( \beta \). Its sort structure makes possible relatively simple proofs of close correspondence between \( \beta \)-reduction and ACCL reduction not possible in Curien’s original theory. More importantly, ACCL has a very natural interpretation in terms of structures and operations commonly used to implement \( \lambda \)-reduction, as well providing a sound framework for entirely new reduction strategies.

In the sequel, we will assume that any \( \lambda \)-terms under consideration are actually terms of the de Bruijn \( \lambda \)-calculus (see Appendix C for a brief review), although we 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.

\[
\begin{align*}
\text{Var} & : \mathcal{L} \quad (\text{variable reference}) \\
\text{Apply}(\mathcal{L}, \mathcal{L}) & : \mathcal{L} \quad (\text{application}) \\
\Lambda(\mathcal{L}) & : \mathcal{L} \quad (\text{abstraction}) \\
[\mathcal{L}, \mathcal{E}] & : \mathcal{L} \quad (\text{closure}) \\
\emptyset : \mathcal{E} & \quad (\text{null environment}) \\
\Diamond : \mathcal{E} & \quad (\text{shift}) \\
(\mathcal{E}, \mathcal{L}) & : \mathcal{E} \quad (\text{expression list}) \\
\mathcal{E} \circ \mathcal{E} & : \mathcal{E} \quad (\text{environment composition})
\end{align*}
\]

The terms of ACCL will be denoted by \( \text{Ter}(ACCL) \) and the closed terms, those terms containing no variables, by \( \text{Ter}_C(ACCL) \).

The following notation (for “de Bruijn” numbers) will be used: ·
Definition 3.2

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

where

\[ \Box^n \equiv \begin{cases} \Box & n = 1 \\ \Box \circ (\Box \circ (\cdots (\Box \circ \Box) \cdots)) & n > 1 \end{cases} \]

n \text{ times} \]

The intuition behind the term structure of \textbf{ACCL} is fairly straightforward: Terms of sort \( \mathcal{L} \) are analogous to terms in the de Bruijn \( \lambda \)-calculus, after variable numbers are encoded as above. Closures are created by the \textbf{ACCL} equivalent of \( \beta \)-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 \textbf{ACCL} as

\[ \langle \langle \langle \cdots \langle \emptyset, M_n \rangle \cdots, M_2 \rangle, M_1 \rangle \rangle. \]

“\( \circ \)” 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:

\begin{align*}
(Beta) & \quad \text{Apply}(\Lambda(A), B) = [A, \langle \emptyset, B \rangle] \\
(AsC) & \quad [[A, E_1], E_2] = [A, E_1 \circ E_2] \\
(NullEL) & \quad \emptyset \circ E = E \\
(NullER) & \quad E \circ \emptyset = E \\
(ShiftE) & \quad \square \circ \langle E, A \rangle = E \\
(VarRef) & \quad [\text{Var}, \langle E, A \rangle] = A \\
(D\Lambda) & \quad [\Lambda(A), E] = \Lambda([[A, \langle E \circ \square, \text{Var} \rangle]]) \\
(DE) & \quad \langle E_1, A \rangle \circ E_2 = \langle E_1 \circ E_2, [A, E_2] \rangle \\
(DApply) & \quad \text{Apply}(A, B), E = \text{Apply}([[A, E], [B, E]]) \\
(AsE) & \quad (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3) \\
(NullC) & \quad [A, \emptyset] = A
\end{align*}

We 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 $\beta$-reduction.

3.3 ACCL as a 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 $\rightarrow_{ACCL}$ will be used to denote the application of a rule of ACCL in some context, i.e., $A \rightarrow_{ACCL} B$ 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 B). We will use similar notation for the subsystem ECCL and applications of single rules of ACCL, e.g. $\rightarrow_{Beta}$. However, we will restrict ourselves in the sequel to the closed terms of ACCL, $\text{Ter}_C(ACCL)$. Since we are interested in using ACCL to model $\lambda$-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 do not hold for arbitrary terms of Curien’s system CCL$\beta$. We will refer to the formal theories and their corresponding rewriting systems by the same name. We first show that ECCL and (Beta) each yield noetherian term-rewriting systems:
3.3.1 Noetherian Subsystems

Theorem 3.1 ECCL is noetherian (strongly normalizing).

Proof We can orient the rules of ECCL using Kamin and Lévy's extended recursive path ordering technique (described in [Hue86, p. 56]).

We first define an ordering $\succeq_o$ on the operators of $\Lambda CCL$ as follows:

$$o \approx_o [\cdot, \cdot] \succ_o (\cdot, \cdot) \succ_o \text{Apply}(\cdot, \cdot) \succ_o \Lambda(\cdot) \succ_o \text{Var} \succ_o \Box \succ_o \emptyset$$

Next, we define $|T|_\Lambda$ to be the number of $\Lambda(\cdot)$ operators in $T$.

Let $A \equiv f(s_1, \ldots, s_m)$ and $B \equiv g(t_1, \ldots, t_n)$ be terms of $\Lambda CCL$ (where $f$ and $g$ are the outermost operators of the terms). We use the definitions above to define a "basic" quasi-ordering $\succeq_t$ on terms as follows:

$$A \succeq_t B$$

if

$$f \succ_o g$$

or

$$f \approx_o g \quad \text{and} \quad |A|_\Lambda \succeq |B|_\Lambda$$

In other words, $\succeq_t$ is the lexicographic combination of the ordering $\succeq_o$ on operators and the number of $\Lambda(\cdot)$ operators in the terms. $\succeq_t$ is clearly a well quasi-ordering.

Finally, following Kamin and Lévy, we extend $\succeq_t$ to a simplification ordering (see [Der87, p. 80]), $\succeq_*$, as follows:

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

if

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

or

$$A \succeq_t B \quad \text{and} \quad A \succeq t_j \quad \text{for all } j = 1 \ldots n$$

or

$$A \approx_t B \quad \text{and} \quad (s_1, \ldots, s_m) \succeq_* (t_1, \ldots, t_n)$$

where $\succeq_*$ is the standard lexicographic extension of $\succeq$ to sequences (see [Der87, p. 97] for details of this extension).

It is easy to show that that $\succeq_t$ and $\succeq_*$ verify the 11 axioms of Kamin and Lévy that must be satisfied to ensure that $\succeq$ is indeed a simplification quasi-ordering. For each $\text{ECCL}$ axiom of the form $L = R$, it is straightforward to show that $L \succeq R$. Since $\succeq$ is a simplification ordering, this implies that if $A \longrightarrow_{\text{ECCL}} B$, $A \succeq B$ and thus that ECCL is noetherian. $\Box$
We also have the following:

**Theorem 3.2 (Beta) is noetherian (strongly normalizing).**

**Proof** Trivial, since each application of (Beta) in a term reduces the number of (Beta) redexes in that term by one. □

### 3.3.2 Normal Forms

It will be useful in the sequel to define several subsets of ACCL terms:

**Definition 3.5** The set of lambda normal forms (LNF) is a subset of the terms of ACCL, defined inductively as follows:

\[
\begin{align*}
\eta & \in \text{LNF} \\
A & \in \text{LNF} \implies \Lambda(A) \in \text{LNF} \\
A & \in \text{LNF}, B \in \text{LNF} \implies \text{Apply}(A, B) \in \text{LNF}
\end{align*}
\]

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

**Theorem 3.3** All lambda-like expressions (terms of sort \( \mathcal{L} \)) of ACCL are reducible to a lambda normal form, using the rules of ECCL, i.e., for all \( A: \mathcal{L} \), there exists \( B \in \text{LNF} \) such that \( A \rightarrow_{ECCL} B \).

**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 a term in LNF is reached, which must happen eventually since ECCL is noetherian. □

For any term \( A: \mathcal{L} \), we will refer to its corresponding term \( B \in \text{LNF} \) by \( \text{lnf}(A) \).

**Definition 3.6** The set of partial environment normal forms (PENF) is a subset of the terms of ACCL defined inductively as follows:

\[
\begin{align*}
\emptyset & \in \text{PENF} \\
\square^n & \in \text{PENF} \\
E & \in \text{PENF} \implies (E, A) \in \text{PENF}
\end{align*}
\]

**Theorem 3.4** All environments (terms of sort \( \mathcal{E} \)) of ACCL are reducible to a partial environment normal form using the rules of ECCL, i.e., for all \( E_1: \mathcal{E} \), there exists \( E_2 \in \text{PENF} \) such that \( E_1 \rightarrow_{ECCL} E_2 \).
Proof Once again, we can observe that every term of sort \( \mathcal{E} \) that is not in \( \text{PENF} \) must contain an ECCL redex. Such redexes can be reduced until the normal form is reached. 

The following two definitions allow us to define the \( \text{ACCL} \) analogue of a weak head normal form (whnf) in the \( \lambda \)-calculus:

**Definition 3.7** The set of abstraction forms \( (AF) \) is a subset of the terms of \( \text{ACCL} \), defined inductively as follows:

\[
\Lambda(A) \in AF \\
A \in AF \implies [A, E] \in AF
\]

**Definition 3.8** The set of weak head normal forms \( (\text{WHNF}) \) is a subset of the terms of \( \text{ACCL} \), defined inductively as follows:

\[
n! \in \text{WHNF} \\
A \in AF \implies A \in \text{WHNF} \\
A \in \text{WHNF}, A \notin AF \implies \text{Apply}(A, B) \in \text{WHNF}
\]

### 3.3.3 Confluence of Subsystems

We are now in a position to show that both \( \text{ECCL} \) and \( (\text{Beta}) \) are confluent subsystems.

In order to show \( \text{ECCL} \) confluent, the following definition and lemma will be required:

**Definition 3.9** The terms \( \text{Null}^i \) are defined by

\[
\text{Null}^0 \equiv \emptyset \\
\text{Null}^i \equiv \{(\cdots(\{\text{Null}^0, i!\}, (i - 1)!), \cdots 1!\}, 0!\} \quad (i > 0)
\]

The terms \( \text{Null}^i \) are behaviorally equivalent to \( \emptyset \), in the sense that for any context \( C[\_]: \mathcal{L} \), \( C[\text{Null}^i] \) is reducible to the same term as \( C[\emptyset] \). In particular, we have the following:

**Lemma 3.1** For all \( A : \mathcal{L} \) and \( n \), there exists \( B \) such that

\[
[A, \text{Null}^n] \rightarrow_{\text{ECCL}} B \leftarrow_{\text{ECCL}} A
\]

12
**Proof** By Theorem 3.3, we know that there exists $A' \in LNF$ such that

$$A \rightarrow_{ECCL} A'$$

We can then show

$$[A', \text{Null}^n] \rightarrow_{ECCL} A'$$

by structural induction on $A'$. Since $A' \in LNF$, there are only three cases:

**Case 1:** $A' \equiv \text{Apply}(B, C)$. We then have

$$[\text{Apply}(B, C), \text{Null}^n] \rightarrow_{ECCL} \text{Apply}([B, \text{Null}^n], [C, \text{Null}^n])$$

and use the induction hypothesis on $[B, \text{Null}^n]$ and $[C, \text{Null}^n]$.

**Case 2:** $A' \equiv \Lambda(B)$. It is then easy to see that

$$[\Lambda(B), \text{Null}^n] \rightarrow_{ECCL} \Lambda([B, \text{Null}^{n+1}])$$

We then use the induction hypothesis on $[B, \text{Null}^{n+1}]$.

**Case 3:** $A' \equiv i!$. We then have

$$[i!, \text{Null}^n] \rightarrow_{ECCL} i!$$

by a simple induction on $n$.

We thus have

$$[A, \text{Null}^n] \rightarrow_{ECCL} [A', \text{Null}^n] \rightarrow_{ECCL} A' \leftarrow_{ECCL} A$$

which completes the proof. □

**Theorem 3.5** ECCL is confluent (thus Church-Rosser) on closed terms, i.e., if $A \rightarrow_{ECCL} B_1$ and $A \rightarrow_{ECCL} B_2$, then there exists $C$ such that $B_1 \rightarrow_{ECCL} C$ and $B_2 \rightarrow_{ECCL} C$.

**Proof** Since ECCL is noetherian (by Theorem 3.1), to show confluence, we need only prove that it is locally confluent. It then suffices to show that each critical pair (see [Hue80] for a formal definition) has a common reduct. This is straightforward, except for the critical pair induced by the rules (DA) and (NullC), for which we must show

$$[A, (\Box, \text{Var})] \rightarrow_{ECCL} B \leftarrow_{ECCL} A$$

for some $B$. This follows from Lemma 3.1 above (since $(\Box, \text{Var}) \equiv \text{Null}^1$). □
We also have the following:

**Theorem 3.6** *(Beta) is confluent, i.e., if \( A \rightarrow_{(\text{Beta})} B_1 \) and \( A \rightarrow_{(\text{Beta})} B_2 \), then there exists \( C \) such that \( B_1 \rightarrow_{(\text{Beta})} C \) and \( B_2 \rightarrow_{(\text{Beta})} C \).

**Proof** *(Beta) has no critical pairs, thus it is trivially locally confluent. Since it is also noetherian (by Theorem 3.2), it is confluent. □

### 3.3.4 ACCL is Confluent

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

**Definition 3.10**

\[ \rightarrow_{\text{Dev}} \equiv \rightarrow_{\text{ECCL}} \cdot \rightarrow_{(\text{Beta})} \cdot \rightarrow_{\text{ECCL}} \]

where ‘\( \cdot \)’ denotes relational composition.

\( \rightarrow_{\text{Dev}} \) is intended to correspond roughly to the notion of a *development* in the \( \lambda \)-calculus. As usual, \( \rightarrow_{\text{Dev}} \) represents the reflexive, transitive closure of \( \rightarrow_{\text{Dev}} \).

First, 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).

The proof of the theorem hinges on the following lemma:

**Lemma 3.2** *(given by the diagram below)*

\[
\begin{array}{c}
\text{ECCL} \\
\downarrow \rho \\
\downarrow \tau \\
\text{ECCL} \\
\end{array} \longleftarrow \longleftarrow \begin{array}{c}
\text{ECCL} \\
\downarrow \rho \\
\downarrow \tau \\
\text{ECCL} \\
\end{array}
\]

Let \( \rho \) and \( \tau \) be the *(Beta) reductions as marked in the diagram above. Then if the redezes in \( \rho \) are disjoint, the redezes in \( \tau \) are also disjoint.

**Proof** See Appendix A.1. □

The following lemma states that *(Beta) and ECCL almost commute, however, the intervention of additional ECCL reductions may be required:
Lemma 3.3 (given by the diagram below)

Let \( \rho \) and \( \tau \) be the (Beta) reductions as marked in the diagram above. Then if the redezes in \( \rho \) are disjoint, the redezes in \( \tau \) are also disjoint.

Proof Follows by noetherian induction (see [Hue80] for a complete exposition of this induction principle) on the left-hand ECCL reduction using Lemma 3.2 and Theorem 3.5. The required construction is given by the following diagram:

The following lemma is also needed:

Lemma 3.4 (given by the diagram below)

Proof The required diagram is constructed as follows:
The next lemma shows that ECCL-(Beta)-ECCL sequences commute:

**Lemma 3.5** *(given by the diagram below)*

**Proof** Follows by noetherian induction on the left-hand ECCL reduction, using Lemmas 3.3 and 3.4 and Theorem 3.5 as base cases. The following construction is required:
We then have:

**Lemma 3.6** \(\rightarrow_{\text{Dev}}\) is confluent on closed terms, i.e.,

\[\text{Diagram:} \]

\[\text{Diagram:} \]

**Proof** The reductions used in Lemma 3.5 are \(\rightarrow_{\text{Dev}}\) contractions, and the theorem thus follows by induction on the lengths of the reductions in the premise. □

**Theorem 3.7** ACCL is confluent on closed terms.

**Proof** \(\rightarrow_{\text{Dev}}\) and \(\rightarrow_{\text{ACCL}}\) are relationally equivalent. Thus from Lemma 3.6, we must conclude that \(\rightarrow_{\text{ACCL}}\) is confluent. □

The principal idea used in the proof above is that one can always find a common reduct for two diverging (Beta) reductions surrounded by ECCL reductions by using reductions of the same sort. Intuitively, new (Beta) redexes can only be created by intervening ECCL reductions. Infinite ACCL reductions can thus only occur when an infinite number of (necessarily) finite ECCL or (Beta) subreductions are alternated.

Theorem 3.7 is a principal result; Curien did not exhibit a confluent system strong enough to model arbitrary reductions in the \(\lambda\)-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:\(\beta\) 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:\(\beta\) on subsets of terms is similar to the confluence proof given here. Lemma 3.3 was used in an earlier version of this paper to prove a somewhat stronger intermediate result than Lemma 3.5; the proof used here was simplified upon observing that Yokouchi's proof of confluence essentially used Lemma 3.3 directly, without resort to a more complicated intermediate lemma. Hardin's proof of confluence relies on the confluence of a subsystem of CCL:\(\beta\) that simulates \(\beta\)-reduction.
Hardin and Yokouchi's proofs of confluence both rely on the fact that a "sub-
stitutive" subset of CCL similar to ECCL is noetherian. This was shown to be
the case by Hardin and Laville [HL86] using an ingenious, but complicated term
ordering. The proof of Theorem 3.1 is considerably simpler.

3.4 Translation

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

Definition 3.11 For any term $M \in \lambda^{DB}$, we can define a corresponding term
$\llbracket M \rrbracket_{ACCL} \in ACCL$ inductively as follows:

$\llbracket i \rrbracket_{ACCL} = i!$

$\llbracket (\lambda N) \rrbracket_{ACCL} = \Lambda(\llbracket N \rrbracket_{ACCL})$

$\llbracket (N_1 N_2) \rrbracket_{ACCL} = \text{Apply}(\llbracket N_1 \rrbracket_{ACCL}, \llbracket N_2 \rrbracket_{ACCL})$

The reverse transformation, $\llbracket \rrbracket_{\lambda}$, is defined in the obvious way on members of
$\text{LNF}$.

3.5 Equivalence

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

Lemma 3.7 Let $M$ and $N$ be arbitrary terms of the (de Bruijn) λ-calculus such
that $M \rightarrow_{\beta} N$. Then

$\llbracket M \rrbracket_{ACCL} \rightarrow_{ACCL} \llbracket N \rrbracket_{ACCL}$

Proof A construction isomorphic to that used by Curien in [Cur86a] to prove a
similar result for CCLβ suffices, and is omitted here. □

The ACCL equivalent of Curien's simulation of $\beta$-reduction in CCLβ has the
following property:

Corollary 3.1 If $M \rightarrow_{\beta} N$, then there exists $B$ such that

$\rho: \llbracket M \rrbracket_{ACCL} \rightarrow_{(\text{Beta})} B$

and

$\sigma: B \rightarrow_{\text{ECCL}} \llbracket N \rrbracket_{ACCL}$
σ effectively carries out the substitution required by a β-contraction. We then have the following:

**Lemma 3.8** Given $A, C \in LNF$ and $B$ such that

$$\rho : A \rightarrow_{(\text{Beta})} B$$

and

$$\sigma : B \rightarrow_{\text{ECCL}} C$$

where the redexes in $\rho$ are disjoint, then

$$\llbracket A \rrbracket_\lambda \rightarrow_{\beta} \llbracket C \rrbracket_\lambda$$

**Proof** Let $R_1, R_2, \ldots, R_n$ be the set of redexes contracted in $\rho$. Consider the set of simulated β-reductions $\gamma_i = \rho_i + \tau_i$ carried out on (Beta) redexes $R_i$ as follows:

$$\rho_i : A_{i-1} \xrightarrow{R_i} (\text{Beta}) B_i$$

$$\tau_i : B_i \xrightarrow{\text{ECCL}} A_i$$

where $A_0 \equiv A$ and $\tau_i$ is the ECCL reduction given by Lemma 3.1 that simulates the substitution operation within $R_i$.

So we have for all $i$

$$\llbracket A_{i-1} \rrbracket_\lambda \rightarrow_{\beta} \llbracket A_i \rrbracket_\lambda$$

Concatenating the reductions gives us

$$\llbracket A \rrbracket_\lambda \rightarrow_{\beta} \llbracket A_n \rrbracket_\lambda$$

Since the reductions $\gamma_i$ are disjoint, we can trivially reorder their components as follows:

Let $\tau = \tau_1 + \tau_2 + \cdots + \tau_n$. We then have

$$\tau : B \rightarrow_{\text{ECCL}} A_n$$

Recalling that $\rho = \rho_1 + \rho_2 + \cdots + \rho_n$, we have

$$\rho : A \rightarrow_{(\text{Beta})} B$$

$$\tau : B \rightarrow_{\text{ECCL}} A_n$$

We now note that the final term $C$ of reduction $\sigma$ in the premise of the lemma and the final term $A_n$ of reduction $\tau$ above are terms in $LNF$. Thus, since ECCL is confluent and $\sigma$ and $\tau$ are coinitial, we must have

$$A_n \equiv C$$
from which we get
\[ [A]_\lambda \rightarrow_\beta [C]_\lambda \]
\[ \square \]

We can now prove the converse of Lemma 3.7:

**Lemma 3.9** Let \( A : \mathcal{L} \rightarrow_{\text{ACCL}} B \). Then
\[ [\text{Inf}(A)]_\lambda \rightarrow_\beta [\text{Inf}(B)]_\lambda \]

**Proof** Let \( A_0 \equiv A \) and \( A_n \equiv B \). Divide the \( \text{ACCL} \) reduction into subreductions alternating (possibly null) ECCL reductions and (Beta) contractions as follows:
\[
A \equiv A_0 \rightarrow_{\text{ECCL}} \hat{A}_0 \rightarrow_{(\text{Beta})} A_1 \rightarrow_{(\text{Beta})} \cdots \rightarrow_{(\text{Beta})} A_{n-1} \rightarrow_{\text{ECCL}} \hat{A}_{n-1} \rightarrow_{(\text{Beta})} A_n \equiv B
\]

Note that since ECCL is confluent,
\[ \text{Inf}(A_i) \equiv \text{Inf}(\hat{A}_i) \]

We can then show that
\[ [\text{Inf}(A)]_\lambda \equiv [\text{Inf}(A_0)]_\lambda \rightarrow_\beta [\text{Inf}(A_1)]_\lambda \rightarrow_\beta \cdots \rightarrow_\beta [\text{Inf}(A_{n-1})]_\lambda \rightarrow_\beta [\text{Inf}(A_n)]_\lambda \equiv [\text{Inf}(B)]_\lambda \]

using Lemma 3.3, Lemma 3.8, and the construction in the following diagram:

\[ \begin{array}{c}
A_0 \equiv A \\
\text{ECCL} \\
\downarrow \\
\text{Inf}(A_0)
\end{array} \quad \begin{array}{c}
\hat{A}_0 \\
\text{(Beta)} \\
\downarrow \\
\text{Inf}(\hat{A}_0)
\end{array} \quad \begin{array}{c}
A_1 \\
\text{ECCL} \\
\downarrow \\
\text{Inf}(A_1)
\end{array} \quad \begin{array}{c}
A_{n-1} \\
\text{ECCL} \\
\downarrow \\
\text{Inf}(A_{n-1})
\end{array} \quad \begin{array}{c}
\hat{A}_{n-1} \\
\text{(Beta)} \\
\downarrow \\
\text{Inf}(\hat{A}_{n-1})
\end{array} \quad \begin{array}{c}
A_n \equiv B \\
\text{ECCL} \\
\downarrow \\
\text{Inf}(A_n)
\end{array}
\]

\[ \begin{array}{c}
\text{Lemma 3.3} \\
\text{Lemma 3.8}
\end{array} \quad \begin{array}{c}
\text{Lemma 3.3} \\
\text{Lemma 3.8}
\end{array}
\]

\[ [\text{Inf}(A_0)]_\lambda \rightarrow_\beta [\text{Inf}(A_1)]_\lambda \rightarrow_\beta \cdots \rightarrow_\beta [\text{Inf}(A_{n-1})]_\lambda \rightarrow_\beta [\text{Inf}(A_n)]_\lambda \equiv [\text{Inf}(B)]_\lambda \]

\[ \square \]

Putting the results from Lemma 3.7 and Lemma 3.9 together yields:
**Theorem 3.8** Given $M \in \text{Ter}(\lambda^{DB})$,

$$M \xrightarrow{\beta} N \text{ if and only if } [M]_{\text{ACCL}} \xrightarrow{\text{ACCL}} [N]_{\text{ACCL}}$$

This result shows that any reduction of a $\text{ACCL}$ term $A \in \text{LNF}$ simulates a reduction in the $\lambda$-calculus.

In [Har89], Hardin proves a similar result for Curien’s Categorical Combinators. However, the proof there is considerably complicated by the presence in CCL of computationally uninteresting terms. Hardin was, however, able to show that in fact such terms do not arise in the process of simulating $\beta$-reduction.

The construction of Lemma 3.9 shows that a single (Beta) contraction can effectively simulate multiple $\beta$-contractions. It is this property, together with the fact that the substitution operation can be deferred except when necessary to yield an outermost redex, that makes an environment-based interpreter an attractive implementation tool for the $\lambda$-calculus.

Any reduction scheme for the $\lambda$-calculus implemented using $\text{ACCL}$ would have to perform $\text{ECCL}$ reductions as well as (Beta) contractions, but it is not unreasonable to count the former as “overhead,” if we ensure that the number of $\text{ECCL}$ reductions required is at worst proportional to the number of (Beta) reductions and the size of the initial term (which is not difficult to do).

### 3.6 Example and Applications

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

$$M \xrightarrow{\beta} \lambda y.y$$

The equivalent term in $\text{ACCL}$ after encoding variables, is given by

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

We then have

$$\Lambda(\text{Apply}(\Lambda(\text{Var}), \text{Var}))$$

$$\xrightarrow{(\text{Beta})} \Lambda([\text{Var}, (\emptyset, \text{Var})])$$

$$\xrightarrow{(\text{VarRef})} \Lambda(\text{Var})$$

and

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

In essence, $\text{ACCL}$ is just a formalization of the informal notions of closure and environment given in the introduction, coupled with a mechanism for indexing environments.
3.7 Term Rewriting and Graph Rewriting

3.7.1 Implementation of Term Rewriting

In order to implement the rules of a term-rewriting system efficiently, it is useful to treat the rules of a term-rewriting system as transformations on trees or, more generally, graphs. We assume that graphs are constructed in the conventional way using nodes and pointers. Thus when a meta-variable X appears on both sides of a rule, instead of copying the entire subtree or subgraph to which X refers when constructing an instance of the right-hand side of the rule, the value of the pointer to the subgraph may be copied instead. Moreover, when a meta-variable is repeated on the right side of the rule, (as with rules (DAppl) and (DE) in ACCL), an instance of the right-hand side of the rule may be constructed in which each pointer corresponding to a repeated variable points to the same subgraph. Thus sharing arises in a natural way.

When rewriting a subgraph S to S', we have the option of redirecting all pointers to S to point to new subgraph S', or overwriting the topmost node of S with the contents of the topmost node of S'. The latter strategy, while generally desirable, is complicated in the presence of projection rules whose right-hand sides contain a single meta-variable: one must either copy the topmost operator of the subgraph to which the right-hand side refers, or use an indirection node of some sort. In the sequel, however, we will be concerned primarily with the efficiency gains made possible through sharing of subgraphs, rather than the details of the sort outlined above.

The natural correspondence between a term-rewriting system and the graphrewriting system induced by the presence of repeated right-hand side pattern variables has been formalized by Barendregt, et.al. [BvEG+87] and Staples [Sta80a, Sta80b,Sta80c,Sta81], among others. Following [BvEG+87], we will refer to graphrewriting systems of this sort as term graph-rewriting systems. We will not, however, pursue further the formal characterizations of term graph-rewriting systems here; an informal approach suffices for the purposes of this discussion.

3.7.2 Graph Rewriting as Parallel Reduction

Contraction of a shared redex can be viewed as parallel contraction of the set of terms which the shared graph represents. The sort of parallel contraction implemented by the term graph-rewriting mechanism outlined above can be described as parallel contraction of identical, disjoint terms, and it is the capabilities and limitations of this sort of parallel contraction with which we will be concerned.

We will use the following notation:
Definition 3.12 The simultaneous contraction of sets of identical, disjoint λACCL redexes is represented by the relation $\longrightarrow_{||\text{ACCL}}$.

Since the redexes contracted by $\longrightarrow_{||\text{ACCL}}$ are assumed to be disjoint, instances of $\longrightarrow_{\text{ACCL}}$ (or $\longrightarrow_{(\text{Beta})}$ or $\longrightarrow_{\text{ECCL}}$) in theorems of Section 3.3 may be uniformly replaced by $\longrightarrow_{||\text{ACCL}}$. This fact will be important in the sequel. Parallel reduction in the $\lambda$-calculus and its relationship with the question of optimality will be discussed in Section 4.2.2.

3.8 A Simple Lambda Interpreter

To illustrate the utility of $\lambda$ACCL both as a notation for specifying manipulations of environments, closures, and $\lambda$-terms and as a vehicle for implementing graph reduction, we present in Figure 1 a simple interpreter for the $\lambda$-calculus: $\text{rwhnf}()$. $\text{rwhnf}()$ is an algorithm that transforms a term of the form $[A, E]$ to the $\lambda$ACCL equivalent of weak head normal form, WHNF (Definition 3.8). It is quite similar to the interpreters of Henderson and Morris [HM76] and Aiello and Prini [AP81]. Figures 2, 3, and 4 consist of algorithms for normalizing environments (to PENF).

The algorithm is specified using rules of $\lambda$ACCL (with the exception of a derived rule, $(\text{Beta}')$, discussed below), a recursive redex selection strategy, and shared terms. The functional notation used in the algorithm should be reasonably self explanatory for someone familiar with a functional 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 $\lambda$ACCL; only those parts of the overwriting term not named by pattern variables are newly allocated. Statements inside “seq...endseq” are executed in sequence. CopyTop($A$) copies the topmost operator of $A$; all of $A$’s subterms are referred to by pointers in the new term.

Our goal is to use $\text{rwhnf}()$ to effectively reduce a term $M \in \text{Tr}(\lambda DB)$ to yield a term $N$ such that

$$M \longrightarrow_{\beta} N \in \text{whnf}$$

if $N$ exists.

$rwhnf()$ is limited to reducing only closure terms, which has the effect of greatly simplifying the interpreter’s structure. This design is accommodated by introducing
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}
      [L_1, E_1]: seq
        C := [L_1, E_1 o E]; {rule (AssC)}
        rwhnf(C)
        endseq;
      (Var: L_1): seq
        rpenf(E); {transform E to PENF}
      case E of
        ∅: C := L_1; {rule (NullC)}
        □^n: skip; {E ≡ □^n, thus C ∈ WHNF}
        (E, A): seq
          rwhnf(A);
          C := CopyTop(A); {rule (VarRef), C ∈ WHNF}
          endseq
        endcase
        endseq
        endcase
        endfn

Figure 1: Algorithm rwhnf()
rp enf (E) ≡
case E of
  ∅: skip;
  □: skip;
  ⟨E₁, A⟩: rpenf(E₁);
  E₁ ⊨ E₂: seq
    rpenf(E₁);
    rpenf(E₂);
    composeEnvs(E)
  endseq
endcase
endfn

Figure 2: Algorithm rpenf()

composeEnvs((E₁ ⊨ E₂): E) ≡
case E₁ of
  ∅: E := E₂;
  □: distribShiftL(E);
  ((□: E₃) ⊨ E₄): seq
    E := E₃ ⊨ ((E₄ ⊨ E₂): E');
    composeEnvs(E');
    distribShiftL(E)
  endseq
  ⟨E₃, A⟩: seq
    E := ⟨(E₃ ⊨ E₂): E', [A, E₂]⟩;
    composeEnvs(E')
  endseq
endcase
endfn

Figure 3: Algorithm composeEnvs()

25
\[ \text{distribShiftL}((\Box: E_1) \circ E_2): E) \equiv \]
\[
\text{case } E_2 \text{ of}
\]
\[
\emptyset: E := E_1; \quad \{ \text{rule (NullER)} \}
\]
\[
\Box^n: \text{skip}; \quad \{ E = \Box^{n+1} \in \text{PENF}, n > 0 \}
\]
\[
\langle E_3, A \rangle: E := E_3 \quad \{ \text{rule (ShiftE)} \}
\]
\text{endcase}
\text{endfn}

Figure 4: Algorithm \text{distribShiftL()}

a modified version of the rule (Beta), (Beta'), which operates directly on redexes
containing closures and is derivable in ACCL:

(Beta') \quad \text{Apply}([\Lambda(A), E], B) = [A, \langle E, B \rangle]

We must also construct an initial term A having the form of a closure:

\[ A \equiv [[M]]_{\text{ACCL}}, \emptyset] \in \text{Ter(ACCL)} \]

A is thus a closure whose term part is the translation of M to ACCL and whose
environment part is the null environment. This has no effect on the outcome of the
reduction since

\[ [A, \emptyset] =_{\text{ACCL}} A \]

Applying \text{rwhnf()} to A then yields \( B \in \text{WHNF} \), from which we can extract \( N \):

\[ N \equiv [[\text{Inf}(B)]_\lambda] \in \text{whnf} \]

Since \text{rwhnf()} simply applies ACCL rules to a term in a fixed order, Theorem
3.8 shows it to be correct, that is, given \( M, N, A, \) and \( B \) as above, we have

\[ \text{rwhnf}(A) = B \in \text{WHNF} \]

implies

\[ M \rightarrow_\beta N \in \text{whnf} \]

The normalization properties of reduction schemes using ACCL depend on
whether or not applications of the rule (Beta) are needed; this property is dis-
cussed in section 4.2.3. rwhnf() can be shown to be weak head normalizing. Though
rwhnf() 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 "lazier" interpreters that can be analyzed using ACCL\textsuperscript{L}.

Another ACCL-based \(\lambda\)-interpreter is described in [Fie90], where it is used to perform incremental \(\lambda\)-reduction.

4 Optimality Criteria

In [Lév78,Lév80], J.-J. Lévy studied the issue of optimal reduction in the \(\lambda\)-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 \(\beta\)-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 \(\lambda\)-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 \(\lambda\)-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, We will review the analysis using labelings here.
4.1 Lévy’s Labeled Lambda Calculus

Lévy’s labeled $\lambda$-calculus was first introduced in [Lév75]. We 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 $\lambda$-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:

$$
\begin{align*}
    l \in S & \implies l \in L \\
    w, v \in L & \implies wv \in L \\
    w \in L & \implies w \in L
\end{align*}
$$

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. $\underline{abcd}$, may occur.

The set of labeled $\lambda$-terms consists of the regular $\lambda$-terms and terms annotated with labels:

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

$$
\begin{align*}
    M \in \text{Ter}(\lambda) & \implies M \in \text{Ter}(\lambda^L) \\
    M \in \text{Ter}(\lambda^L), w \in L & \implies (M^w) \in \text{Ter}(\lambda^L)
\end{align*}
$$

where $w$ 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. We 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, $\epsilon$. We define label concatenation and underlining to behave on the null label as follows:

$$\epsilon w = w$$
$$we = w$$
$$\epsilon = \epsilon$$

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

The $\beta$-contraction rule is now defined for labeled terms as follows:

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

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

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

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

Though the labeled $\beta$-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 ($w$), is an indication that the term was effectively generated by contraction of a redex having degree $w$ (this assumes, as we 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 $\text{Ter}(\lambda^L)$ allow multiple labelings of parenthesized terms, which can be created as a result of labeled $\beta$-contraction. This requires an auxiliary reduction rule for labels:

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

$$C[ (M^w)^v ] \rightarrow_{\text{lab}} C[ M^{vw} ]$$

where $C$ is an arbitrary context and $M^w$ is a term of $\text{Ter}(\lambda^L)$.
We then have:

**Definition 4.5** Labeled \( \beta \)-reduction, \( \rightarrow_{\beta L} \), is the reflexive, transitive closure of ( \( \rightarrow_{\text{lab}} \cup \rightarrow_{\beta L} \) ), where 'U' 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, we will assume it has been simplified as much as possible using \( \rightarrow_{\beta L} \). This assumption is technically justified by the following theorem:

**Theorem 4.1** ([Lév75]) \( \rightarrow_{\beta L} \) has the Church-Rosser (confluence) property, i.e., if \( M \rightarrow_{\beta L} N_1 \) and \( M \rightarrow_{\beta L} N_2 \), then there exists \( P \) such that \( N_1 \rightarrow_{\beta L} P \) and \( N_2 \rightarrow_{\beta L} P \).

Thus labeled \( \lambda \)-reduction is as “well-behaved” as its unlabeled counterpart, and, in a sense, is a strict refinement of the regular \( \lambda \)-reduction. Ignoring the labels, it is simply regular \( \lambda \)-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 \( \text{Ter}(\lambda^L) \). Then the erasure of \( M^l \), \( \text{Er}(M^l) \) is the same term with all the labels erased.

**Definition 4.7** Let \( M \) be a term of \( \text{Ter}(\lambda) \). Then \( M^l \in \text{Ter}(\lambda^L) \) is a labeling of \( M \) if and only if \( \text{Er}(M^l) = M \).

We can also define the erasure of a reduction (overloading the meaning of ‘Er’):

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

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

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

4.2.1 Labels and Residuals

With the machinery of the labeled $\lambda$-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

$$\rho: M \rightarrow_{\beta} N$$

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

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

be the labeled version of $\rho$. Then a redex $R_j$ in any term of $\rho$ (not necessarily a redex contracted by $\rho$) is a member of family class $F_w$ if and only if the corresponding redex $R_j^l$ in $\rho^l$ 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 $\lambda$-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**

$$\sigma''_1: (\lambda y. (yy))(Iz) \rightarrow_{\beta} (Iz)(Iz) \rightarrow_{\parallel \beta} zz$$

where ' $\rightarrow_{\parallel \beta}$ ' represents parallel $\beta$-contraction. Note that parallel $\beta$-contraction subsumes ordinary 0 or 1 step $\beta$-reduction ( $\rightarrow_{\equiv \beta}$ ), which is a development of 0 or 1 redexes.
Parallel reductions are represented thus:

\[ \sigma: M_0 \xrightarrow{C_1}_\beta M_1 \xrightarrow{C_2}_\beta \cdots \xrightarrow{C_n}_\beta M_n \]

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

\[ \sigma: M_0 \xrightarrow{F_{w_1}} M_1 \xrightarrow{F_{w_2}} \cdots \xrightarrow{F_{w_n}} M_n \]

is family-complete if and only if 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 \( M \in \text{Ter}(\lambda) \) is needed if and only if, for all terminating reductions \( \sigma \) with initial term \( M \), either \( R \) or one of its residuals is contracted in \( \sigma \).

**Definition 4.13 (Lévy)** A parallel reduction

\[ \rho: M_0 \xrightarrow{C_1}_\beta M_1 \xrightarrow{C_2}_\beta \cdots \xrightarrow{C_n}_\beta M_n \]

is a call-by-need reduction if and only if there is at least one needed redex in each \( C_i \).

The leftmost redex in a term is always needed, although there will often be more than one needed redex in a hterm (see [BKK87] for a detailed analysis of the phenomenon of neededness).
4.2.4 Optimality Theorem

We will consider here various classes of parallel reductions. A reduction strategy that contracts redexes from a given class may produce shorter reductions of a given term than may be possible in another class of reductions. This leads to the following definition:

**Definition 4.14** A reduction strategy $S$ is **optimal** for a class of parallel reductions $C$ if the number of (parallel) contractions required in the reduction of an arbitrary term $T$ using strategy $S$ is less than or equal to the number of contractions required by any reduction of $T$ in class $C$.

Thus, for instance, $S$ may at each step contract sets of redexes from a class larger than $C$.

Wadsworth's graph reduction technique allowed members of some (but not all) residual classes (i.e., sets of residuals of some redex in a previous intermediate term of the reduction) to be shared and thus contracted simultaneously. We define such reductions as follows:

**Definition 4.15** A **parallel reduction**

$$\rho: M \longrightarrow_{\parallel} N$$

is residual-parallel if the set of redexes contracted at each step consists only of members of some residual class.

Wadsworth's technique made possible reductions that are shorter in many cases than reductions on conventional terms. The question then arises as to whether parallel reductions of large enough sets of redexes (through sharing) are optimal. Lévy's optimality theorem answers this question in the affirmative:

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

$$\rho: M \longrightarrow_{\parallel} N$$

is optimal for the class of residual-parallel reductions if

- $\rho$ is family-complete.
- $\rho$ is call-by-need.
Since reductions performed on non-shared terms are degenerate parallel reductions of members of residual classes, a family-complete, call-by-need reduction is also optimal for the class of all non-parallel reductions, that is, traditional term reduction.

Note that the theorem does not require that an optimal strategy always use shared redexes—if all family classes have one member, a family-complete reduction requires contraction of only one redex at a time. However, in general, families will have more than one member and a practical implementation that allows unit-time parallel contraction of a complete family will require some form of sharing.

Since the members of a family class may consist of different substitution instances of the same term, it is essential that a practical reduction scheme allow such sets of differing terms to be effectively shared. ACCL provides a means to do this: the closure. Two closures with different environment subterms (representing different free variable substitutions) may share a common subterm: e.g., \([A, E_1]\) and \([A, E_2]\), where the two instances of \(A\) are shared. This idea is exploited in Example 2.5.

5 Labeled ACCL

To make a precise connection between redex families in the \(\lambda\)-calculus and shared terms with closures and environments in ACCL, we will define a labeled variant of ACCL, ACCL\(^L\). ACCL\(^L\) is intended to be analogous to Lévy’s labeled \(\lambda\)-calculus.

**Definition 5.1** The axioms of ACCL\(^L\) are as follows:

\[
\begin{align*}
(Beta) & \quad \text{Apply}(\Lambda(A)^u, B)^u = [A^u, \langle \emptyset, B^u \rangle] \\
(AsSC) & \quad [[A, E_1]^u, E_2]^v = [A^u, E_1 \circ E_2] \\
(NullEL) & \quad \emptyset \circ E = E \\
(NullER) & \quad E \circ \emptyset = E \\
(ShiftE) & \quad \Box \circ \langle E, A \rangle = E \\
(VarRef) & \quad [Var^u, \langle E, A \rangle]^v = A^u \\
(DA) & \quad \Lambda(A)^v, E^v = \Lambda([A, \langle E \circ \Box, \text{Var} \rangle])^u \\
(DE) & \quad (E_1, A)^v \circ E_2 = (E_1 \circ E_2, [A, [E_2]]) \\
(DAApply) & \quad \text{Apply}(A, B)^u, E)^v = \text{Apply}([A, E], [B, E])^u \\
(AsSE) & \quad (E_1 \circ E_2) \circ E_3 = E_1 \circ (E_2 \circ E_3) \\
(NullC) & \quad [A^u, \emptyset]^v = A^u \\
(DLabel) & \quad [A, E]^u = [A^u, E]
\end{align*}
\]

Note that the (DLabel) has no analogue in unlabeled ACCL. It is the ACCL\(^L\) equivalent of the the convention allowing the removal of parentheses in multiply-labeled parenthesized terms of \(\lambda^L\). By analogy with the labeled \(\lambda\)-calculus, the
degree of a labeled (Beta) redex is the label of its abstraction term; e.g.,

\[ \text{Apply}(\Lambda(A^w, B)^x) \]

has degree \( w \). As with the labeled-lambda calculus, we can define the erasure of labeled ACCL\(^L\) terms and reductions, yielding their unlabeled counterparts. Likewise, we can define the lifting of unlabeled terms and reductions to labeled versions.

The translations \( \llbracket \cdot \rrbracket_{\lambda L} \) and \( \llbracket \cdot \rrbracket_{\text{ACCL} L} \) are defined in the obvious way analogous to their unlabeled counterparts. Theorems 3.7, 3.3, 3.9, and 3.8 all apply to ACCL\(^L\) and \( \lambda L \); the proofs are analogous and are omitted.

We also wish to consider parallel labeled reductions, in which sets of identical (including labeling) terms are contracted at each step. The parallel analogues of the aforementioned theorems are straightforward to define. We will make use in particular of the following refinement of Lemma 3.9:

**Lemma 3.9'** Let \( A : L \) be a term of ACCL\(^L\) such that

\[ \rho : A \rightarrow_{\llbracket \cdot \rrbracket_{\text{ACCL} L}} B \]

Let \( R_i \) be the set of redexes contracted in the \( i \)th parallel (Beta) contraction of \( \rho \), \( w_i \) be the degree of the redexes contracted in \( R_i \), and \( L_{\rho} \) be the multiset consisting of the labels \( v_i \).

Then there exists reduction \( \sigma \) such that

\[ \sigma : \llbracket \text{Inf}(A) \rrbracket_{\lambda L} \rightarrow_{\llbracket \cdot \rrbracket_{\beta L}} \llbracket \text{Inf}(B) \rrbracket_{\lambda L} \]

where all the members of the set \( S_j \) of redexes contracted in the \( j \)th parallel \( \beta \)-contraction of \( \sigma \) have the same degree, \( v_j \).

Furthermore, if we let \( L_{\sigma} \) be the multiset consisting of the labels \( v_j \), then we have

\[ L_{\rho} = L_{\sigma} \]

**Proof** Since the redexes in each parallel (Beta) contraction of \( \rho \) are by definition disjoint, we may use a construction analogous to that used in the proof of Lemma 3.9 to yield \( \sigma \), where the reduction of disjoint \( \beta \)-redexes at each stage of the construction is replaced by a single parallel \( \beta \)-contraction.

We note that the degrees of the (Beta) redexes in the premise of the labeled analogue of Lemma 3.3 are preserved in the degrees of the (Beta) redexes in its conclusion. From this we can conclude that \( L_{\rho} = L_{\sigma} \). □

We can then make the following definitions:
Definition 5.2 Let A contain (Beta) redex R and let $\rho$ be the reduction

$$\rho: A \rightarrow_{\text{ACCL}} B$$

Let $A^l$ be a labeling of $A$ such that $R$ has degree $w$ and all other subterms of $A^l$ have the null label. Finally, let $\rho^l$ be the labeled counterpart of $\rho$ such that

$$\rho^l: A^l \rightarrow_{\text{ACCL}^l} B^l$$

Then (Beta) redex $S$ in $B$ is a $\lambda$-residual of $R$ (relative to reduction $\rho$) if its labeled counterpart $S^l$ in $B^l$ has degree $w$.

If $R$ is a (Beta) redex in $A: \mathcal{L}$, we will refer without ambiguity to the residuals of $R$ in $\text{Inf}(A)$, since $\text{ACCL}^l$ is confluent and any ECCL reduction of $A$ to $\text{Inf}(A)$ must yield the same set of residuals.

Definition 5.3 Let $\rho$ be the following parallel reduction of $A: \mathcal{L}$:

$$\rho: A \rightarrow_{\|\text{ACCL}} B$$

Then $\rho$ is $\lambda$-optimal if the number of (parallel) (Beta) contractions in $\rho$ is less than or equal the number of parallel $\beta$-contractions in an optimal (in the sense of Lévy) $\lambda$-reduction from $\|\text{Inf}(A)\|_\lambda$ to $\|\text{Inf}(B)\|_\lambda$.

Definition 5.4 A (Beta) redex $R$ in a term $A: \mathcal{L}$ is $\lambda$-needed if and only if some $\lambda$-residual of $R$ in $\text{Inf}(A)$ is needed in the sense of Definition 4.12.

We can now apply Lévy's optimality criteria directly to reductions in ACCL, using $\text{ACCL}^l$. The idea is to consider each (Beta) contraction in a term $A$ as representing a parallel $\beta$-contraction on the corresponding $\lambda$-term $\|\text{Inf}(A)\|_\lambda$.

Theorem 5.1 Let $\rho$ be the following parallel reduction of $A: \mathcal{L}$:

$$\rho: A \rightarrow_{\|\text{ACCL}} B$$

Let $A^l \in \text{ACCL}^l$ be a labeling of $A$ such that all of $A$'s subterms have unique labels, and let $\rho^l$ be the labeled counterpart of $\rho$ such that

$$\rho^l: A^l \rightarrow_{\|\text{ACCL}^l} B^l$$

Let $S_i$ be the set of (identical) (Beta) redexes contracted in the $i$th parallel (Beta) contraction in $\rho$ and $w_i$ be the degree of the redexes contracted in by the labeled counterpart of $S_i$ in $\rho^l$, $S^l_i$. Then the reduction $\rho$ is $\lambda$-optimal only if
— For all \( i, j \), \( w_i \neq w_j \).

— For all \( i \), some redex in \( R_i \) is \( \lambda \)-needed.

**Proof** Consider the reduction \( \rho^1 \) constructed in Lemma 3.9' such that

\[
\rho^1 : [\text{Inf}(A^i)]_{\lambda_L} \rightarrow_{\beta} [\text{Inf}(B^i)]_{\lambda_L}
\]

If a redex with degree \( w \) is contracted twice in \( \rho \), a \( \beta \)-redex with degree \( w \) must be contracted at least twice in \( \rho^1 \) (since the multisets of redex degrees in the two reductions are equal). But then there must be some shorter family-complete parallel \( \beta \)-reduction \( \sigma \) such that

\[
\sigma : [\text{Inf}(A^i)]_{\lambda_L} \rightarrow_{\parallel \beta} [\text{Inf}(B^i)]_{\lambda_L}
\]

in which the redex labeled \( w \) is reduced only once (since \( \sigma \) is family-complete), thus \( \rho \) is not \( \lambda \)-optimal. Similarly, if \( \rho \) contracts an unneeded \( \beta \)-redex, then an unneeded \( \beta \)-redex is contracted in \( \rho^1 \), and there is a shorter reduction in which no unneeded \( \beta \)-redex is contracted. \( \square \)

If we construct a \( \lambda \)-interpreter whose action can be expressed in terms of some application of the rules of \( \Lambda \text{CCL} \), 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

We can now show that there is no \( \lambda \)-optimal reduction possible in the term graph-rewriting system corresponding to \( \Lambda \text{CCL} \) in the sense discussed in Section 3.7, and formalized in [BvEG+87]. We do so by exhibiting a \( \lambda \)-term \( Q \) for which every \( \Lambda \text{CCL} \) term graph-reduction causes more than one (Beta) redex to be contracted in the corresponding labeled form. The term \( Q \) is as follows:

\[
(\lambda x.((x A^d)(x B^e)))(\lambda y.((\lambda z.(z^2t)(z^b u))(\lambda w.y^c v)))
\]

where \( A \) and \( B \) are arbitrary \( \lambda \)-abstractions. Not all subterms are given labels for the sake of clarity.

We will not enumerate all possible reductions of \( Q \)'s corresponding \( \Lambda \text{CCL}^L \) 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 \( Q \) if no prior (Beta) redexes with the same label are to be reduced twice:

\[
(((\bullet, \langle y := A^d \rangle) : C_1)(\bullet, \langle y := B^e \rangle) : C_2))
\]

As before, \( \langle \cdot \rangle \) represents a ACCL environment with bound variables 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:

\[
(((\bullet, \langle y := A^d \rangle) : C_1)(\bullet, \langle y := B^e \rangle) : C_2)
\]

which reduces to

\[
(((A^{de}v)(A^{de}v)) \bullet, \langle y := B^e \rangle) : C_2))
\]

in which two redexes of the form \( (A^{de}v) \) 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:

\[
(((\bullet, \langle z := \lambda w. (A^{de}v) \rangle) : C_1')(\bullet, \langle z := \lambda w. (B^{ec}v) \rangle) : C_2')
\]

which reduces to

\[
((\lambda w. \bullet)^{at}((\lambda w. \bullet)^{bt})((\lambda w. \bullet)^{at})((\lambda w. \bullet)^{bt})
\]

\( (A^{de}v) \quad (B^{ec}v) \)

The term above has two (actually, two sets) of unshared redexes with the same degree, e.g., \( ((\lambda w. (A^{de}v))^{at}) \) and \( ((\lambda w. (B^{ec}v))^{at}) \). 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 term graph-rewriting implementations of closures and environments are inadequate for implementing 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

A system almost identical to ACCL has been independently proposed by Abadi, et.al. [ACCL90]. 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.

There have been several proposed implementations of optimal λ-reduction: by Staples [Sta82] and more recently by Lamping [Lam90] and Kathail [Kat90]. These schemes seem to allow more terms to be shared than are possible using traditional environment or substitution mechanisms. However, they are notable for their extreme complexity, and it is not clear that the overhead incurred by these schemes in order to ensure that family classes are always shared is not prohibitive. Such complexity may be inherent, however, in the peculiar nature of redex families.

8 Conclusions

We have chosen here to investigate the limitations of term graph-rewriting implementations of ACCL. An alternative (and perhaps slightly more precise) formulation of the problem could be made by first defining the notion of ACCL redex families, beginning, e.g., with the idea of residual for term-rewriting systems defined by [Klo80]. We could then show that no ACCL-family-complete reduction yields a corresponding family-complete reduction in the λ-calculus, as well as show that term graph-rewriting in fact implements ACCL-family-complete reductions.

We note from the counterexample of Section 6 that sharing of redex families requires that existing sharing in a graph be respected during the process of substitution, in addition to being created as a a consequence of the application of rewrite rules with multiple instances of meta-variables. In order to implement optimal reduction, it thus seems that more powerful rewriting systems (i.e., non-left-linear systems that can “test” for equality of subterms) or graph rewriting systems with capabilities beyond those of term graph-rewriting are required.

In summary, we have described a new formal system, ACCL, with which one can describe a wide variety of reduction methods for the λ-calculus using environments, closures, and shared terms. We have shown that terms and reductions in ACCL correspond in a natural to their counterparts in the λ-calculus; λ-reduction schemes
using ACCL may thus be proved correct trivially. However, by making the "micro-manipulations" required to implement substitution explicit, ACCL makes evident a broad range of options for implementation of efficient interpreters. We have also described a labeled variant of ACCL, ACCL\textsuperscript{L}, which can be used as a precise tool to analyze environment and closure-based implementations of the \(\lambda\)-calculus to determine the extent to which the implementation is lazy. We have shown, however, that the standard term graph-rewriting implementation of ACCL is insufficient for implementing optimal reduction schemes.

9 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 Thérèse Hardin and Jean-Jacques Lévy for fruitful conversation, providing helpful comments, and pointing me toward related work.

A Supplementary Proofs

A.1 Proof of Lemma 3.2

In this section, we give the proof for Lemma 3.2, which will require some preliminary lemmas and definitions.

We first define two subsystems of ECCL that will be useful in the sequel.

Definition A.1 The axioms of FCCL consist of those of ECCL without rule (DApply).

Definition A.2 The axiom \((\text{VarRef'})\) is as follows:

\[ [\text{Var}, \langle E_1, [A, E_2]\rangle] = [A, E_2] \]

Note that \((\text{VarRef'})\) is simply \((\text{VarRef})\) restricted to a smaller set of terms.

Definition A.3 The axioms of GCCL are as follows:

\((\text{AssC}), (\text{DE}), (\text{AssE}), (\text{ShiftE}), (\text{NullER}), (\text{VarRef'})\)

Using these new systems, we now obtain the following results:
Lemma A.1 FCCL and (Beta) commute, i.e.,

Proof FCCL and (Beta) have no critical pairs and therefore commute weakly. Since both are noetherian, the result follows by noetherian induction. □

Lemma A.2 Let GCCL$^{-1}$ be the rewrite system obtained by orienting the equations of GCCL from right to left. Then GCCL$^{-1}$ and (Beta) commute, i.e.,

Proof GCCL$^{-1}$ and (Beta) have no critical pairs, thus commute weakly. The result follows by noetherian induction on the (Beta) reduction. □

Lemma A.3 GCCL and (Beta) reductions may be permuted, that is, the following diagram holds:

Proof Simply reverse the arrows of the GCCL$^{-1}$ reductions used in Lemma A.2 to obtain GCCL reductions and the desired result (this technique was suggested by Proposition 5.5 of [Klo80, p. 46]). □

We are now in a position to prove Lemma 3.2:

Lemma 3.2 (given by the diagram below)
Let \( \rho \) and \( \tau \) be the \((\text{Beta})\) reductions as marked in the diagram above. Then if the redezexes in \( \rho \) are disjoint, the redezexes in \( \tau \) are also disjoint.

**Proof** Let \( \sigma \) and \( \rho \) be the ECCL contraction and \((\text{Beta})\) reduction, respectively, in the premise of the lemma as shown in the diagram above.

We first note that if neither \( \sigma \) nor any of its residuals (relative to \( \rho \)) creates an instance of a critical pair with one of the redezexes in \( \rho \), then the diagram can be closed trivially (using residuals), which must also be disjoint if the \((\text{Beta})\) redezexes in \( \rho \) are disjoint. Otherwise, the only rule that can create an instance of a critical pair is \((\text{DAapply})\).

Since rule \((\text{Beta})\) can create no new \((\text{Beta})\) redezexes, the redezexes in \( \rho \) can be permuted arbitrarily. Thus if \( \sigma \) is an application of rule \((\text{DAapply})\) that creates an instance of a critical pair with some relox in \( \rho \), we can reorder the redezexes in \( \rho \) so that the contraction of the overlapping relox occurs first.

Therefore, we assume the following without loss of generality:

1. \( \sigma \) is an application of rule \((\text{DAapply})\).
2. There is an instance of a critical pair between the first relox contracted in \( \rho \) and \( \sigma \).

Given the assumptions above, we can construct the following diagram using previous lemmas.
The square in the diagram marked ‘*’ is easy to construct using rules of the appropriate sort starting from the critical pair of terms created by an overlap of rules (DApplay) and (Beta). By treating rules of FCCL and GCCL as ECCL rules, we obtain the required diagram. We finally note that the constructions in ‘*’, Lemma A.1, and A.3 each yield (Beta) reductions in their conclusions with disjoint redexes if the redexes in the (Beta) reductions of their premises are also disjoint. □

B  The Lambda Calculus and Term Rewriting Systems

This section contains a brief review of selected terminology from term-rewriting systems and λ-calculus used herein. The conventions will generally follow those of [Bar84], to which the reader is referred for details, although a few are taken from [Klo80, Section 5], [Hue80], or [BKK87].

B.1 Notation and Terminology

C[M] denotes a context containing M, i.e., C[M] is a 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 →_β.

The reflexive, transitive closure of →_β, β-reduction, is denoted by →_β.

Other notions of reduction for term-rewriting systems will be defined using analogous notation: if →_R is a reduction relation, then →_R will denote its reflexive, transitive closure, and =_R the induced equivalence.

→_R and →_S commute weakly if for all X, Y, and Z such that X →_R Y and X →_S Z, there exists W such that Y →_S W and Z →_R W.

→_R and →_S commute if for all X, Y, and Z such that X →_R Y and X →_S Z, there exists W such that Y →_S W and Z →_R W.

→_R is locally confluent if →_R commutes weakly with itself.

→_R is confluent (or Church-Rosser) if →_R commutes with itself.

→_R is noetherian (or strongly normalizing) if there is no infinite sequence of the form

X_1 →_R X_2 →_R ⋯ →_R X_n →_R ⋯

Since ‘=’ will be reserved to represent equality induced by a reduction relation, we will use ‘≡’ to denote syntactic identity of terms. In the case of the λ-calculus,
we will identify on the syntactic level terms that are identical modulo changes of bound variable and avoid the machinery of $\alpha$-conversion, i.e., we will feel free to say

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

$M$ is a normal form (or $M \in \text{nf}$) if and only if it contains no redexes.

$M$ is a head normal form ($M \in \text{hnf}$) if and only if it has the form

$$\lambda x_1 \cdots x_n. (y P_1 \cdots P_m), \quad n, m \geq 0$$

where $y$ and the $x_i$ are arbitrary variables and the $P_j$ are arbitrary $\lambda$-terms.

$M$ is a weak head normal form ($M \in \text{whnf}$) if and only if it has either of the forms

$$\lambda x. N$$

or

$$(x P_1 \cdots P_n), \quad n \geq 0$$

for arbitrary variable $x$ and arbitrary $\lambda$-term $N$ and terms $P_i$.

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

$$\sigma: M_0 \xrightarrow{P_1}_R M_1 \xrightarrow{P_2}_R M_2 \xrightarrow{P_3}_R \cdots \xrightarrow{P_n}_R M_n$$

$\sigma$ designates the the entire reduction sequence. The $M_i$ are the terms of the reduction. The $P_i$ denote the redexes contracted at each step. Where clear from context, the $P_i$ may be omitted. Occasionally, it will be convenient to elide the intermediate terms and denote the entire sequence by $\sigma: M_0 \longrightarrow_R M_n$.

C The de Bruijn Lambda Calculus

The de Bruijn $\lambda$-calculus [dB72,dB78] is a variant of the $\lambda$-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 $\lambda$-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 $\lambda$-calculus eliminates the need for $\alpha$-conversion. The following definitions are from [Cur86a]
Definition C.1 The set of terms in the de Bruijn $\lambda$-calculus, designated $\text{Ter}(\lambda^{DB})$, is defined inductively as follows

\[ n \in \mathcal{N} \implies n \in \text{Ter}(\lambda^{DB}) \]
\[ M, N \in \text{Ter}(\lambda^{DB}) \implies (MN) \in \text{Ter}(\lambda^{DB}) \]
\[ M \in \text{Ter}(\lambda^{DB}) \implies \lambda M \in \text{Ter}(\lambda^{DB}) \]

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

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

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

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

Substitution, $\beta$-reduction, and $\eta$-reduction can be suitably redefined on $\lambda^{DB}$ such that $M \rightarrow_{\beta, \eta} N$ if and only if $M_{DB} \rightarrow_{\beta, \eta} N_{DB}$. For a concise exposition of the details of $\beta$-reduction and the substitution process, see [Cur86a]

References


