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, **ΛCCL**, based on Curien's *Categorical Combinators* [Cur86a]. We show that **ΛCCL** has properties that make it especially suitable for analysis and implementation of a wide range of λ-reduction schemes using shared environments, closures, or λ-terms. In particular, the term structure of **ΛCCL** is very closely related to the structure of existing abstract machines for λ-reduction. **ΛCCL** is powerful enough to mimic arbitrary (strong) reduction in the λ-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 λ-calculus, whose correctness follows trivially from the properties of **ΛCCL**.

We then describe a *labeled* variant of **ΛCCL**, **ΛCCL**$^{\mathrm{L}}$, which can be used as a tool to determine the degree of "laziness" possessed by various λ-reduction schemes. In particular, **ΛCCL**$^{\mathrm{L}}$ 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. However, using **ΛCCL**$^{\mathrm{L}}$, we show that the sharing allowed by environments and closures in **ΛCCL** 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 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. Unfortunately, the exact nature of laziness in various implementation 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 λ-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 λ-calculus [Chu41,Bar84,HS86], the de Bruijn λ-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 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

$$\begin{aligned}
\sigma_1 &: M \longrightarrow (Iz)(Iz) \longrightarrow z(Iz) \longrightarrow zz \\
\sigma_2 &: M \longrightarrow (Iz)(Iz) \longrightarrow (Iz)z \longrightarrow zz \\
\sigma_3 &: M \longrightarrow (\lambda y.(yy))z \longrightarrow zz
\end{aligned}$$

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

$$\sigma'_1: (\lambda y.(yy))(Iz) \longrightarrow (\bullet\,\bullet)_{(Iz)} \longrightarrow (\bullet\,\bullet)_{z} \equiv zz$$

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

$$\begin{array}{l}\rho_1\colon N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (Iw)(\bullet z) \longrightarrow w(\bullet z)\\ \qquad\qquad\qquad\quad \lambda y.(Iy) \quad\;\; \lambda y.(Iy) \qquad\; \lambda y.(Iy)\\ \qquad \longrightarrow w(Iz) \longrightarrow wz\end{array}$$

$$\begin{array}{l}\rho_2\colon N \longrightarrow (\bullet w)(\bullet z) \longrightarrow (\bullet w)(\bullet z) \longrightarrow w(\bullet z) \longrightarrow wz\\ \qquad\qquad\qquad\quad \lambda y.(Iy) \qquad\quad \lambda y.y \qquad \lambda y.y\end{array}$$

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

$$\begin{aligned}
(\lambda y.(yy))(Iz) &\longrightarrow [(yy),\ \langle\!\langle y := (Iz) \rangle\!\rangle] \\
&\longrightarrow ([y,\ \bullet][y,\ \bullet]) \\
&\qquad\qquad\quad \langle\!\langle y := (Iz) \rangle\!\rangle \\
&\longrightarrow ([y,\ \bullet][y,\ \bullet]) \cdots \longrightarrow (zz) \\
&\qquad\qquad\quad \langle\!\langle y := z \rangle\!\rangle
\end{aligned}$$

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 $N \equiv (N_1N_2)$, where $N_1 \equiv \lambda x.(xw)(xz)$ and $N_2 \equiv \lambda y.(Iy)$. Then, using shared environments, we have:

Example 2.5

$$
\begin{aligned}
N &\longrightarrow [(xw)(xz),\ \langle\!\langle x := N_2 \rangle\!\rangle] \\
&\twoheadrightarrow (([x,\ \bullet][w,\ \bullet])\ [(xz),\ \bullet]) \\
&\qquad\qquad\qquad\qquad\qquad \langle\!\langle x := N_2 \rangle\!\rangle \\
&\longrightarrow ((\bullet\ [w,\ \bullet])\ [(xz),\ \bullet]) \\
&\qquad\qquad\qquad\qquad \langle\!\langle x := \bullet \rangle\!\rangle \\
&\qquad\qquad\qquad\qquad\qquad \lambda y.(Iy) \\
&\longrightarrow (([\bullet,\ \langle\!\langle y := [w,\ \bullet] \rangle\!\rangle])\ [(xz),\ \bullet]) \\
&\qquad\qquad\qquad\qquad\qquad \langle\!\langle x := \lambda y.\ \bullet \rangle\!\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad (Iy) \\
&\longrightarrow (([\bullet,\ \langle\!\langle y := [w,\ \bullet] \rangle\!\rangle])\ [(xz),\ \bullet]) \\
&\qquad\qquad\qquad\qquad\qquad \langle\!\langle x := \lambda y.\ \bullet \rangle\!\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad y \\
&\longrightarrow \cdots
\end{aligned}
$$

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 confluent 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 and analyzing λ-reduction using environments, we will define a new 2-sorted equational theory, **ΛCCL**, akin to Curien's theory **CCLβ**. Its sort structure makes possible relatively simple proofs of close correspondence between β-reduction and **ΛCCL** reduction not possible in Curien's original theory. More importantly, **ΛCCL** has a very natural interpretation in terms of structures and operations commonly used to implement λ-reduction, as well providing a sound framework for entirely new reduction strategies.

In the sequel, we will assume that any λ-terms under consideration are actually terms of the de Bruijn λ-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* **ΛCCL** *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.

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

The terms of **ΛCCL** will be denoted by *Ter*(**ΛCCL**) and the *closed terms*, those terms containing no variables, by Ter_C(**ΛCCL**).

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

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

The intuition behind the term structure of **ΛCCL** 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 **ΛCCL** 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 **ΛCCL** 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* **ΛCCL** *are as follows:*

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

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

Definition 3.4 *The axioms of* **ECCL** *are those of* **ΛCCL** *without rule* **(Beta)**.

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

3.3 ΛCCL as a Rewriting System on Closed Terms

By orienting the equations of **ΛCCL** from left to right, they can be treated as a term-rewriting system. The notation $\longrightarrow_{\Lambda\mathrm{CCL}}$ will be used to denote the application of a rule of **ΛCCL** in some context, i.e., $A \longrightarrow_{\Lambda\mathrm{CCL}} B$ if and only if $A \equiv C[X]$, X may be rewritten to Y using one of the oriented axioms of **ΛCCL**, 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 **ΛCCL**, e.g. $\longrightarrow_{(\mathbf{Beta})}$, However, we will restrict ourselves in the sequel to the *closed terms* of **ΛCCL**, $\mathit{Ter}_C(\mathbf{\Lambda CCL})$. Since we are interested in using **ΛCCL** 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 **ΛCCL**, the restriction to closed terms makes it possible to prove properties of **ΛCCL** that did not hold for arbitrary terms of Curien's system **CCLβ**. 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 $\mathbf{\Lambda CCL}$ as follows:

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

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 $\mathbf{\Lambda CCL}$ (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

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

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

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

or

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

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 **ECCL** axiom of the form $L = R$, it is straightforward to show that $L \succ R$. Since $\succ$ is a simplification ordering, this implies that if $A \longrightarrow_{\mathrm{ECCL}} B$, $A \succ B$ 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 **ΛCCL** terms:

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

$$\begin{array}{rcl} n! \in LNF & & \\ A \in LNF & \Longrightarrow & \Lambda(A) \in LNF \\ A \in LNF, B \in LNF & \Longrightarrow & \mathrm{Apply}(A,\, B) \in LNF \end{array}$$

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* **ΛCCL** *are reducible to a lambda normal form, using the rules of* **ECCL**, *i.e., for all $A{:}\,\mathcal{L}$, there exists $B \in LNF$ such that $A \longrightarrow\!\!\!\!\!\rightarrow_{\mathrm{ECCL}} B$.*

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 $B \in LNF$ by $lnf(A)$.

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

$$\begin{array}{rcl} \emptyset \in PENF & & \\ \square^n \in PENF & & \\ E \in PENF & \Longrightarrow & \langle E,\, A\rangle \in PENF \end{array}$$

Theorem 3.4 *All environments (terms of sort $\mathcal{E}$) of* **ΛCCL** *are reducible to a partial environment normal form using the rules of* **ECCL**, *i.e., for all $E_1{:}\,\mathcal{E}$, there exists $E_2 \in PENF$ such that $E_1 \longrightarrow\!\!\!\!\!\rightarrow_{\mathrm{ECCL}} E_2$.*

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. □

The following two definitions allow us to define the **ΛCCL** analogue of a weak head normal form (*whnf*) in the λ-calculus:

Definition 3.7 *The set of* abstraction forms *(AF) is a subset of the terms of* **ΛCCL**, *defined inductively as follows:*

$$\begin{array}{rcl} \Lambda(A) \in AF & & \\ A \in AF & \Longrightarrow & [A,\, E] \in AF \end{array}$$

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

$$\begin{array}{rcl} n! \in \mathit{WHNF} & & \\ A \in AF & \Longrightarrow & A \in \mathit{WHNF} \\ A \in \mathit{WHNF}, A \notin AF & \Longrightarrow & \mathrm{Apply}(A,\, B) \in \mathit{WHNF} \end{array}$$

3.3.3 Confluence of Subsystems

We are now in a position to show that both **ECCL** and **(Beta)** are confluent subsystems.

In order to show **ECCL** confluent, the following definition and lemma will be required:

Definition 3.9 *The terms* Null^i *are defined by*

$$\begin{array}{rcl} \mathrm{Null}^0 & \equiv & \emptyset \\ \mathrm{Null}^i & \equiv & \langle\langle\cdots\langle\langle\Box^i,\, i!\rangle,\, (i-1)!\rangle, \cdots 1!\rangle,\, 0!\rangle \quad (i > 0) \end{array}$$

The terms Null^i are behaviorally equivalent to $\emptyset$, in the sense that for any context $C[\ \ ]: \mathcal{L}$, $C[\mathrm{Null}^i]$ 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*

$$[A,\, \mathrm{Null}^n] \twoheadrightarrow_{\mathrm{ECCL}} B \twoheadleftarrow_{\mathrm{ECCL}} A$$

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

$$A \longrightarrow\!\!\!\!\!\rightarrow_{\text{ECCL}} A'$$

We can then show

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

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

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

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

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

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

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

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

by a simple induction on n.

We thus have

$$[A,\ \text{Null}^n] \quad \longrightarrow\!\!\!\!\!\rightarrow_{\text{ECCL}} \quad [A',\ \text{Null}^n] \quad \longrightarrow\!\!\!\!\!\rightarrow_{\text{ECCL}} \quad A' \quad \leftarrow\!\!\!\!\!\leftarrow\!\!\!-_{\text{ECCL}} \quad A$$

which completes the proof. □

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

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 **(DΛ)** and **(NullC)**, for which we must show

$$[A,\ \langle\square,\ \text{Var}\rangle] \longrightarrow\!\!\!\!\!\rightarrow_{\text{ECCL}} B \leftarrow\!\!\!\!\!\leftarrow\!\!\!-_{\text{ECCL}} A$$

for some B. This follows from Lemma 3.1 above (since $\langle\square,\ \text{Var}\rangle \equiv \text{Null}^1$). □

We also have the following:

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

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 ΛCCL is Confluent

We can now show **ΛCCL** 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 3.10

$$\longrightarrow_{\mathbf{Dev}} \;\equiv\; \twoheadrightarrow_{\mathbf{ECCL}} \cdot \twoheadrightarrow_{(\mathbf{Beta})} \cdot \twoheadrightarrow_{\mathbf{ECCL}}$$

where '·' denotes relational composition.

$\longrightarrow_{\mathbf{Dev}}$ is intended to correspond roughly to the notion of a *development* in the λ-calculus. As usual, $\twoheadrightarrow_{\mathbf{Dev}}$ represents the reflexive, transitive closure of $\longrightarrow_{\mathbf{Dev}}$.

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)*

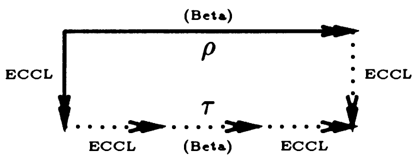
$$\begin{tikzcd}
\cdot \arrow[rrr, two heads, "(\mathrm{Beta})", "\rho"'] \arrow[d, "\mathrm{ECCL}"'] & & & \cdot \arrow[d, dotted, two heads, "\mathrm{ECCL}"] \\
\cdot \arrow[r, dotted, two heads, "\mathrm{ECCL}"'] & \cdot \arrow[r, dotted, two heads, "(\mathrm{Beta})"', "\tau"] & \cdot \arrow[r, dotted, two heads, "\mathrm{ECCL}"'] & \cdot
\end{tikzcd}$$

Let ρ *and* τ *be the* **(Beta)** *reductions as marked in the diagram above. Then if the redexes in* ρ *are disjoint, the redexes in* τ *are also disjoint.*

Proof See Appendix A.1. □

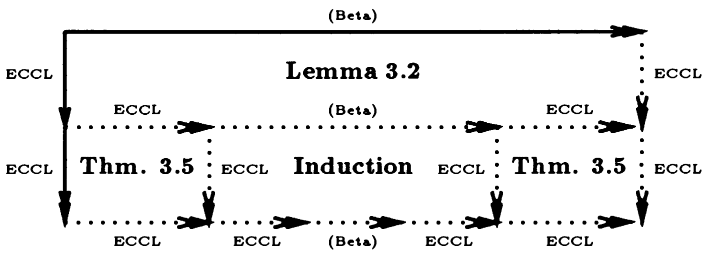
The following lemma states that **(Beta)** and **ECCL** *almost* commute, however, the intervention of additional **ECCL** reductions may be required:

Lemma 3.3 *(given by the diagram below)*



Let ρ *and* τ *be the* **(Beta)** *reductions as marked in the diagram above. Then if the redexes in* ρ *are disjoint, the redexes in* τ *are also disjoint.*

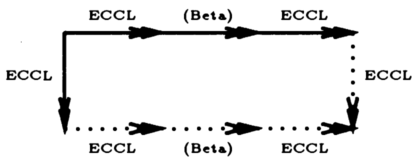
Proof Follows by noetherian induction (see [Hue80] for a complete exposition of this induction principle) on the left-hand **ECCL** reduction using Lemma 3.2 and Theorem 3.5. The required construction is given by the following diagram:



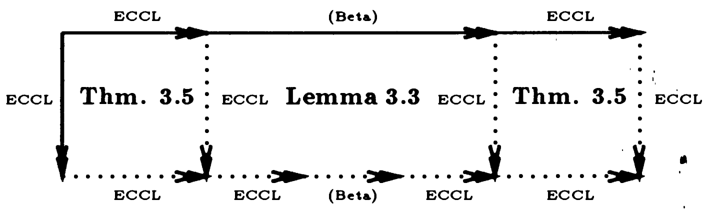
□

The following lemma is also needed:

Lemma 3.4 *(given by the diagram below)*



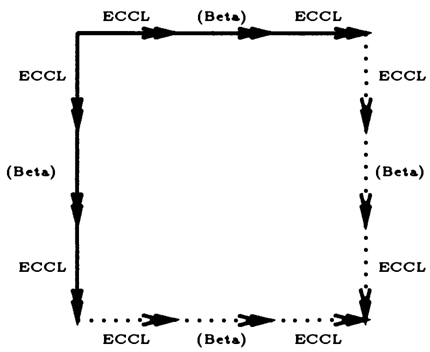
Proof The required diagram is constructed as follows:



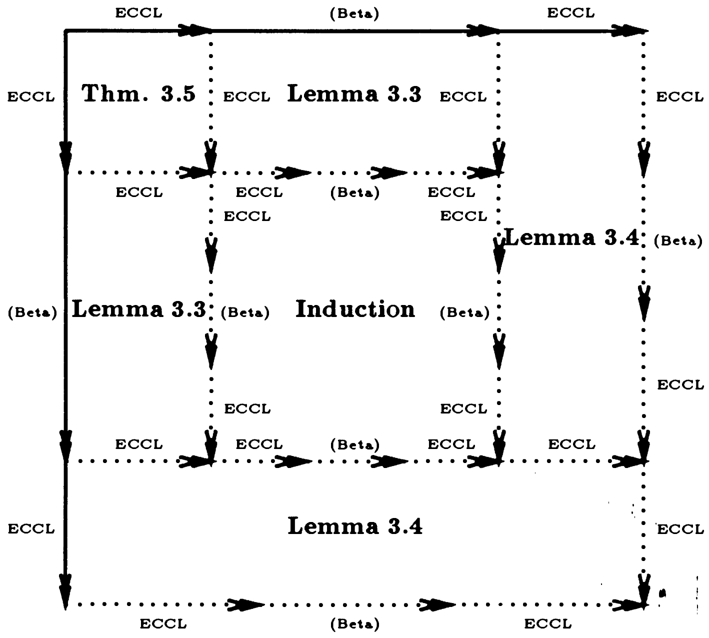
□

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

$$\begin{tikzcd} \cdot \arrow[r, two heads, "\mathrm{Dev}"] \arrow[d, two heads, "\mathrm{Dev}"'] & \cdot \arrow[d, dotted, two heads, "\mathrm{Dev}"] \\ \cdot \arrow[r, dotted, two heads, "\mathrm{Dev}"'] & \cdot \end{tikzcd}$$

Proof The reductions used in Lemma 3.5 are $\longrightarrow_{\mathrm{Dev}}$ contractions, and the theorem thus follows by induction on the lengths of the reductions in the premise. □

Theorem 3.7 $\mathbf{\Lambda CCL}$ *is confluent on closed terms.*

Proof $\twoheadrightarrow_{\mathrm{Dev}}$ and $\twoheadrightarrow_{\Lambda\mathrm{CCL}}$ are relationally equivalent. Thus from Lemma 3.6, we must conclude that $\twoheadrightarrow_{\Lambda\mathrm{CCL}}$ 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 **ΛCCL** 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 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 **ΛCCL** 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 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**β that simulates β-reduction.

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] 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 **ΛCCL**.

Definition 3.11 *For any term $M \in \lambda^{\mathrm{DB}}$, we can define a corresponding term $[\![M]\!]_{\Lambda\mathrm{CCL}} \in$ **ΛCCL** inductively as follows:*

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

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

3.5 Equivalence

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

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

$$[\![M]\!]_{\Lambda\mathrm{CCL}} \twoheadrightarrow_{\Lambda\mathrm{CCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

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

The **ΛCCL** equivalent of Curien's simulation of β-reduction in **CCLβ** has the following property:

Corollary 3.1 *If $M \longrightarrow_\beta N$, then there exists B such that*

$$\rho\colon [\![M]\!]_{\Lambda\mathrm{CCL}} \longrightarrow_{(\mathbf{Beta})} B$$

and

$$\sigma\colon B \twoheadrightarrow_{\mathrm{ECCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

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*

$$\rho: A \twoheadrightarrow_{(\mathbf{Beta})} B$$

and

$$\sigma: B \twoheadrightarrow_{\mathrm{ECCL}} C$$

where the redexes in ρ are disjoint, then

$$[\![A]\!]_\lambda \twoheadrightarrow_\beta [\![C]\!]_\lambda$$

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

$$\begin{array}{llll} \rho_i: & A_{i-1} & \xrightarrow{R_i}_{(\mathbf{Beta})} & B_i \\ \tau_i: & B_i & \twoheadrightarrow_{\mathrm{ECCL}} & A_i \end{array}$$

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

So we have for all i

$$[\![A_{i-1}]\!]_\lambda \longrightarrow_\beta [\![A_i]\!]_\lambda$$

Concatenating the reductions gives us

$$[\![A]\!]_\lambda \twoheadrightarrow_\beta [\![A_n]\!]_\lambda$$

Since the reductions γ_i are disjoint, we can trivially reorder their components as follows:

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

$$\tau: B \twoheadrightarrow_{\mathrm{ECCL}} A_n$$

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

$$\begin{array}{llll} \rho: & A & \twoheadrightarrow_{(\mathbf{Beta})} & B \\ \tau: & B & \twoheadrightarrow_{\mathrm{ECCL}} & A_n \end{array}$$

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

$$A_n \equiv C$$

from which we get

$$[\![A]\!]_\lambda \twoheadrightarrow_\beta [\![C]\!]_\lambda$$

□

We can now prove the converse of Lemma 3.7:

Lemma 3.9 *Let* $A : \mathcal{L} \twoheadrightarrow_{\Lambda\mathrm{CCL}} B$. *Then*

$$[\![\mathit{Inf}(A)]\!]_\lambda \twoheadrightarrow_\beta [\![\mathit{Inf}(B)]\!]_\lambda$$

Proof Let $A_0 \equiv A$ and $A_n \equiv B$. Divide the **ΛCCL** reduction into subreductions alternating (possibly null) **ECCL** reductions and **(Beta)** contractions as follows:

$$\begin{array}{l} A \equiv A_0 \twoheadrightarrow_{\mathrm{ECCL}} \hat{A}_0 \longrightarrow_{\textbf{(Beta)}} A_1 \longrightarrow_{\textbf{(Beta)}} \cdots \\ \quad \longrightarrow_{\textbf{(Beta)}} A_{n-1} \twoheadrightarrow_{\mathrm{ECCL}} \hat{A}_{n-1} \longrightarrow_{\textbf{(Beta)}} A_n \equiv B \end{array}$$

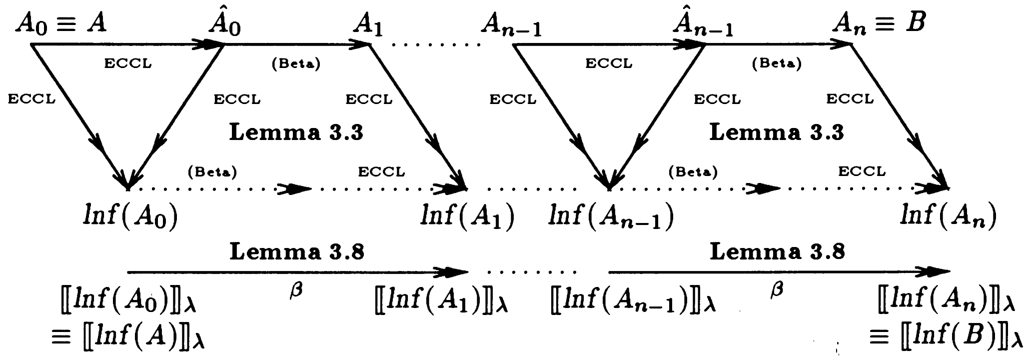
Note that since **ECCL** is confluent,

$$\mathit{Inf}(A_i) \equiv \mathit{Inf}(\hat{A}_i)$$

We can then show that

$$\begin{array}{l} [\![\mathit{Inf}(A)]\!]_\lambda \equiv [\![\mathit{Inf}(A_0)]\!]_\lambda \twoheadrightarrow_\beta [\![\mathit{Inf}(A_1)]\!]_\lambda \twoheadrightarrow_\beta \cdots \\ \quad \twoheadrightarrow_\beta [\![\mathit{Inf}(A_{n-1})]\!]_\lambda \twoheadrightarrow_\beta [\![\mathit{Inf}(A_n)]\!]_\lambda \equiv [\![\mathit{Inf}(B)]\!]_\lambda \end{array}$$

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



□

Putting the results from Lemma 3.7 and Lemma 3.9 together yields:

Theorem 3.8 *Given* $M \in Ter(\lambda^{\mathrm{DB}})$,

$$M \longrightarrow\!\!\!\!\rightarrow_\beta N \text{ if and only if } [\![M]\!]_{\Lambda\mathrm{CCL}} \longrightarrow\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}} [\![N]\!]_{\Lambda\mathrm{CCL}}$$

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

The construction of Lemma 3.9 shows that a single **(Beta)** contraction can effectively simulate multiple β-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 λ-calculus.

Any reduction scheme for the λ-calculus implemented using **ΛCCL** would have to perform **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 **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

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

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

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

We then have

$$\begin{aligned} &\Lambda(\mathrm{Apply}(\Lambda(\mathrm{Var}),\ \mathrm{Var})) \\ &\qquad \longrightarrow_{(\mathbf{Beta})} \Lambda([\mathrm{Var},\ \langle\emptyset,\ \mathrm{Var}\rangle]) \\ &\qquad \longrightarrow_{(\mathbf{VarRef})} \Lambda(\mathrm{Var}) \end{aligned}$$

and

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

In essence, **ΛCCL** 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 **(DApply)** and **(DE)** in **ΛCCL**), 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 graph-rewriting 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 graph-rewriting 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* $\mathbf{\Lambda CCL}$ *redexes is represented by the relation* $\longrightarrow_{\|\Lambda\mathrm{CCL}}$.

Since the redexes contracted by $\longrightarrow_{\|\Lambda\mathrm{CCL}}$ are assumed to be disjoint, instances of $\twoheadrightarrow_{\Lambda\mathrm{CCL}}$ (or $\twoheadrightarrow_{(\mathbf{Beta})}$ or $\twoheadrightarrow_{\mathrm{ECCL}}$) in theorems of Section 3.3 may be uniformly replaced by $\twoheadrightarrow_{\|\Lambda\mathrm{CCL}}$. This fact will be important in the sequel. Parallel reduction in the λ-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 $\mathbf{\Lambda CCL}$ both as a notation for specifying manipulations of environments, closures, and λ-terms and as a vehicle for implementing graph reduction, we present in Figure 1 a simple interpreter for the λ-calculus: *rwhnf*(). *rwhnf*() is an algorithm that transforms a term of the form $[A, E]$ to the $\mathbf{\Lambda CCL}$ 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 $\mathbf{\Lambda CCL}$ (with the exception of a derived rule, **(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 $\mathbf{\Lambda CCL}$; 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 *rwhnf*() to effectively reduce a term $M \in Ter(\lambda^{\mathrm{DB}})$ to yield a term N such that

$$M \twoheadrightarrow_{\beta} N \in whnf$$

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 accomodated 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}
        [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 := CopyTop(A);                       {rule (VarRef), C ∈ WHNF}
                    endseq
            endcase
            endseq
    endcase
endfn
```

Figure 1: Algorithm *rwhnf*()

$rpenf(E) \equiv$
    case E **of**
        $\emptyset$: **skip**; $\{E \in PENF\}$
        $\square^n$: **skip**; $\{E \in PENF\}$
        $\langle E_1, A\rangle$: $rpenf(E_1)$;
        $E_1 \circ E_2$: **seq**
            $rpenf(E_1)$;
            $rpenf(E_2)$;
            $composeEnvs(E)$
            endseq
    endcase
endfn

Figure 2: Algorithm *rpenf*()

$composeEnvs((E_1 \circ E_2): E) \equiv$ $\{E_1, E_2 \in PENF\}$
    case E_1 **of**
        $\emptyset$: $E := E_2$; $\{rule$ **(NullEL)**$\}$
        $\square$: $distribShiftL(E)$;
        $((\square: E_3) \circ E_4)$: **seq**
            $E := E_3 \circ ((E_4 \circ E_2): E')$; $\{rule$ **(AssE)**$\}$
            $composeEnvs(E')$;
            $distribShiftL(E)$
            endseq
        $\langle E_3, A\rangle$: **seq**
            $E := \langle (E_3 \circ E_2): E', [A, E_2]\rangle$; $\{rule$ **(DE)**$\}$
            $composeEnvs(E')$
            endseq
    endcase
endfn

Figure 3: Algorithm *composeEnvs*()

$$
\begin{array}{ll}
\textit{distribShiftL}(((\Box\colon E_1)\circ E_2)\colon E) \equiv & \{E_2 \in \textit{PENF}\} \\
\quad \textbf{case } E_2 \textbf{ of} & \\
\qquad \emptyset\colon\ E := E_1; & \{\textit{rule } (\textbf{NullER})\} \\
\qquad \Box^n\colon\ \textbf{skip}; & \{E = \Box^{n+1} \in \textit{PENF}, n > 0\} \\
\qquad \langle E_3,\ A\rangle\colon\ E := E_3 & \{\textit{rule } (\textbf{ShiftE})\} \\
\quad \textbf{endcase} & \\
\textbf{endfn} &
\end{array}
$$

Figure 4: Algorithm *distribShiftL*()

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

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

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

$$A \equiv [[\![M]\!]_{\Lambda\text{CCL}},\ \emptyset] \in \mathit{Ter}(\mathbf{\Lambda CCL})$$

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

$$[A,\ \emptyset] =_{\Lambda\text{CCL}} A$$

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

$$N \equiv [\![\mathit{lnf}(B)]\!]_\lambda \in \mathit{whnf}$$

Since *rwhnf*() simply applies **ΛCCL** 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

$$\mathit{rwhnf}(A) = B \in \mathit{WHNF}$$

implies

$$M \twoheadrightarrow_\beta N \in \mathit{whnf}$$

The normalization properties of reduction schemes using **ΛCCL** depend on whether or not applications of the rule **(Beta)** are *needed*; this property is discussed 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 $\mathbf{\Lambda CCL}$, and functions as a starting point for much "lazier" interpreters that can be analyzed using $\mathbf{\Lambda CCL^{L}}$.

Another $\mathbf{\Lambda CCL}$-based λ-interpreter is described in [Fie90], where it is used to perform *incremental* λ-reduction.

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, We will review the analysis using labelings here.

4.1 Lévy's Labeled Lambda Calculus

Lévy's *labeled λ-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 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:*

$$
\begin{aligned}
l \in S &\implies l \in \mathrm{L} \\
w, v \in \mathrm{L} &\implies wv \in \mathrm{L} \\
w \in \mathrm{L} &\implies \underline{w} \in \mathrm{L}
\end{aligned}
$$

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. $\underline{a\underline{bc}d}$, 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* $\mathrm{Ter}(\lambda^{\mathrm{L}})$, *is defined as follows:*

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

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*, ϵ. We define label concatenation and underlining to behave on the null label as follows:

$$\begin{aligned} \epsilon w &= w \\ w\epsilon &= w \\ \underline{\epsilon} &= \epsilon \end{aligned}$$

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* $\mathrm{Ter}(\lambda^{\mathrm{L}})$ *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 $\mathrm{Ter}(\lambda^{\mathrm{L}})$.

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 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 $Ter(\lambda^{\mathrm{L}})$ 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* $\mathrm{Ter}(\lambda^{\mathrm{L}})$*:*

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

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

We then have:

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

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

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* $\text{Ter}(\lambda^{\text{L}})$*. 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* $M^l \in \text{Ter}(\lambda^{\text{L}})$ *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* σ^l *be a labeled reduction. Then the* erasure *of* σ^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* 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* $\text{Ter}(\lambda)$, M^l *be some labeling of* M*, and* $\sigma\colon M \twoheadrightarrow_{\beta} N$*. Then the lifted reduction* $\mathit{Lift}(\sigma, M^l)$ *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*

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

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 if and only if 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

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

where '$\longrightarrow_{||\beta}$' represents parallel β-contraction. Note that parallel β-contraction subsumes ordinary 0 or 1 step β-reduction ($\longrightarrow_{\equiv\beta}$), 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*

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

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 $M \in \mathrm{Ter}(\lambda)$ is* needed *if and only if, 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*

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

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 [BKKS87] 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*

$$\rho\colon M \twoheadrightarrow_{\|\beta} N$$

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*

$$\rho\colon M \twoheadrightarrow_{\|\beta} N$$

is optimal *for the class of residual-parallel reductions if*

— ρ *is* family-complete.

— ρ *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. **ΛCCL** 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 ΛCCL

To make a precise connection between redex families in the λ-calculus and shared terms with closures and environments in **ΛCCL**, we will define a *labeled* variant of **ΛCCL**, $\mathbf{\Lambda CCL^L}$. $\mathbf{\Lambda CCL^L}$ is intended to be analogous to Lévy's labeled λ-calculus.

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

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

Note that the **(DLabel)** has no analogue in unlabeled **ΛCCL**. It is the $\mathbf{\Lambda CCL^L}$ 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.,

$$\text{Apply}(\Lambda(A^u)^w,\ B^v)^z$$

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

The translations $[\![\cdot]\!]_{\lambda^\mathrm{L}}$ and $[\![\cdot]\!]_{\Lambda\mathrm{CCL^L}}$ are defined in the obvious way analogous to their unlabeled counterparts. Theorems 3.7, 3.3, 3.9, and 3.8 all apply to $\mathbf{ACCL^L}$ and λ^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* $A\colon \mathcal{C}$ *be a term of* $\mathbf{ACCL^L}$ *such that*

$$\rho\colon A \longrightarrow\!\!\!\!\rightarrow_{||\Lambda\mathrm{CCL^L}} B$$

Let R_i *be the set of redexes contracted in the ith parallel* **(Beta)** *contraction of* ρ, w_i *be the degree of the redexes contracted in* R_i, *and* L_ρ *be the multiset consisting of the labels* w_i.

Then there exists reduction σ *such that*

$$\sigma\colon [\![\mathit{lnf}(A)]\!]_{\lambda^\mathrm{L}} \longrightarrow\!\!\!\!\rightarrow_{||\beta^\mathrm{L}} [\![\mathit{lnf}(B)]\!]_{\lambda^\mathrm{L}}$$

where all the members of the set S_j *of redexes contracted in the jth parallel* β-*contraction of* σ *have the same degree*, v_j.

Furthermore, if we let L_σ *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 ρ are by definition disjoint, we may use a construction analogous to that used in the proof of Lemma 3.9 to yield σ, where the reduction of disjoint β-redexes at each stage of the construction is replaced by a single parallel β-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 ρ be the reduction*

$$\rho : A \longrightarrow\!\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}} B$$

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 ρ^l be the labeled counterpart of ρ such that

$$\rho^l : A^l \longrightarrow\!\!\!\!\!\rightarrow_{\Lambda\mathrm{CCL}^{\mathrm{L}}} B^l$$

Then (**Beta**) *redex S in B is a* λ-residual *of R (relative to reduction ρ) 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 $\mathit{lnf}(A)$, since $\mathbf{\Lambda CCL^L}$ is confluent and any **ECCL** reduction of A to $\mathit{lnf}(A)$ must yield the same set of residuals.

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

$$\rho : A \longrightarrow\!\!\!\!\!\rightarrow_{\|\Lambda\mathrm{CCL}} B$$

Then ρ is λ-optimal *if the number of (parallel)* (**Beta**) *contractions in ρ is less than or equal the number of parallel β-contractions in an optimal (in the sense of Lévy) λ-reduction from $[\![\mathit{lnf}(A)]\!]_\lambda$ to $[\![\mathit{lnf}(B)]\!]_\lambda$.*

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

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

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

$$\rho : A \longrightarrow\!\!\!\!\!\rightarrow_{\|\Lambda\mathrm{CCL}} B$$

Let $A^l \in \mathbf{\Lambda CCL^L}$ be a labeling of A such that all of A's subterms have unique labels, and let ρ^l be the labeled counterpart of ρ such that

$$\rho^l : A^l \longrightarrow\!\!\!\!\!\rightarrow_{\|\Lambda\mathrm{CCL}^{\mathrm{L}}} B^l$$

Let S_i be the set of (identical) (**Beta**) *redexes contracted in the ith parallel* (**Beta**) *contraction in ρ and w_i be the degree of the redexes contracted in by the labeled counterpart of S_i in ρ^l, S_i^l. Then the reduction ρ is λ-optimal only if*

— *For all* i, j, $w_i \neq w_j$.

— *For all* i, *some redex in* R_i *is* λ*-needed.*

Proof Consider the reduction $\hat{\rho}^l$ constructed in Lemma 3.9′ such that

$$\hat{\rho}^l\colon [\![lnf(A^l)]\!]_{\lambda\mathrm{L}} \twoheadrightarrow_\beta [\![lnf(B^l)]\!]_{\lambda\mathrm{L}}$$

If a redex with degree w is contracted twice in ρ, a β-redex with degree w must be contracted at least twice in $\hat{\rho}^l$ (since the multisets of redex degrees in the two reductions are are equal). But then there must be some *shorter* family-complete parallel β-reduction σ such that

$$\sigma\colon [\![lnf(A^l)]\!]_{\lambda\mathrm{L}} \twoheadrightarrow_{\|\beta} [\![lnf(B^l)]\!]_{\lambda\mathrm{L}}$$

in which the redex labeled w is reduced only once (since σ is family-complete), thus ρ is not λ-optimal. Similarly, if ρ contracts an unneeded β-redex, then an unneeded β-redex is contracted in $\hat{\rho}^l$, and there is a shorter reduction in which no unneeded β-redex is contracted. □

If we construct a λ-interpreter whose action can be expressed in terms of some application of the rules of **ΛCCL**, 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* λ-optimal reduction possible in the term graph-rewriting system corresponding to **ΛCCL** in the sense discussed in Section 3.7, and formalized in [BvEG+87]. We do so by exhibiting a λ-term Q for which *every* **ΛCCL** term graph-reduction causes more than one **(Beta)** redex to be contracted in the corresponding labeled form. The term Q is as follows:

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

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

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

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

As before, $\langle\!\langle\cdot\rangle\!\rangle$ represents a **ACCL** environment with bound variables indicated explicitly. The notation $T\colon N$ 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:

$$\begin{array}{l} (([\bullet,\ \langle\!\langle y := A^{d} \rangle\!\rangle]\colon C_1)([\bullet,\ \langle\!\langle y := B^{e} \rangle\!\rangle]\colon C_2)) \\ \qquad\qquad ((y^{c}v)(y^{c}v)) \end{array}$$

which reduces to

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

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:

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

which reduces to

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

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 $((\lambda w.(B^{ec}v))^{a}t)$. 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 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 **ΛCCL** has been independently proposed by Abadi, et.al. [ACCL90]. Its term structure is isomorphic to that of **ΛCCL**, 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 **ΛCCL**. An alternative (and perhaps slightly more precise) formulation of the problem could be made by first defining the notion of **ΛCCL** redex families, beginning, e.g., with the idea of residual for term-rewriting systems defined by [Klo80]. We could then show that no **ΛCCL**-family-complete reduction yields a corresponding family-complete reduction in the λ-calculus, as well as show that term graph-rewriting in fact implements **ΛCCL**-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, **ΛCCL**, 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 **ΛCCL** 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**, $\mathbf{ACCL}^{\mathrm{L}}$, which can be used as a precise tool to analyze environment and closure-based implementations of the λ-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* **(VarRef′)** *is as follows:*

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

Note that **(VarRef′)** is simply **(VarRef)** restricted to a smaller set of terms.

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

(AssC), **(DE)**, **(AssE)**, **(ShiftE)**, **(NullER)**, **(VarRef′)**

Using these new systems, we now obtain the following results:

Lemma A.1 **FCCL** *and* **(Beta)** *commute, i.e.,*

$$\begin{tikzcd}
\cdot \arrow[r, two heads, "\text{(Beta)}"] \arrow[d, two heads, "\text{FCCL}"'] & \cdot \arrow[d, two heads, dotted, "\text{FCCL}"] \\
\cdot \arrow[r, two heads, dotted, "\text{(Beta)}"'] & \cdot
\end{tikzcd}$$

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

$$\begin{tikzcd}
\cdot \arrow[r, two heads, "\text{(Beta)}"] \arrow[d, two heads, "\text{GCCL}^{-1}"'] & \cdot \arrow[d, two heads, dotted, "\text{GCCL}^{-1}"] \\
\cdot \arrow[r, two heads, dotted, "\text{(Beta)}"'] & \cdot
\end{tikzcd}$$

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

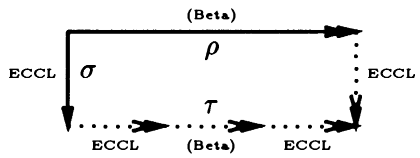
Lemma A.3 **GCCL** *and* **(Beta)** *reductions may be* permuted, *that is, the following diagram holds:*

$$\begin{tikzcd}
& \cdot \arrow[dr, two heads, "\text{(Beta)}"] & \\
\cdot \arrow[ur, two heads, "\text{GCCL}"] \arrow[dr, two heads, dotted, "\text{GCCL}"'] & & \cdot \\
& \cdot \arrow[ur, two heads, dotted, "\text{(Beta)}"'] &
\end{tikzcd}$$

Proof Simply reverse the arrows of the $\mathbf{GCCL}^{-1}$ 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 ρ and τ be the **(Beta)** *reductions as marked in the diagram above. Then if the redexes in ρ are disjoint, the redexes in τ are also disjoint.*

Proof Let σ and ρ be the **ECCL** contraction and **(Beta)** reduction, respectively, in the premise of the lemma as shown in the diagram above.

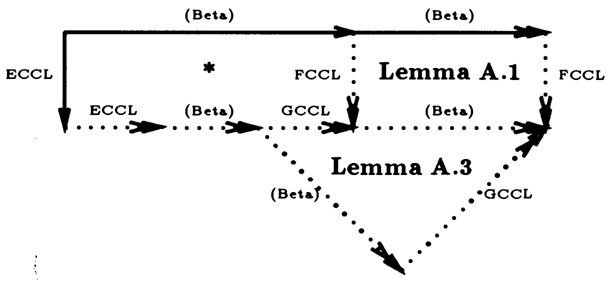
We first note that if neither σ nor any of its residuals (relative to ρ) creates an instance of a critical pair with one of the redexes in ρ, then the diagram can be closed trivially (using residuals), which must also be disjoint if the **(Beta)** redexes in ρ are disjoint. Otherwise, the only rule that can create an instance of a critical pair is **(DApply)**.

Since rule **(Beta)** can create no new **(Beta)** redexes, the redexes in ρ can be permuted arbitrarily. Thus if σ is an application of rule **(DApply)** that creates an instance of a critical pair with some redex in ρ, we can reorder the redexes in ρ so that the contraction of the overlapping redex occurs first.

Therefore, we assume the following without loss of generality:

1. σ is an application of rule **(DApply)**.
2. There is an instance of a critical pair between the first redex contracted in ρ and σ.

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 **(DApply)** 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 [BKKS87].

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 $\longrightarrow_\beta$.

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

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

$\longrightarrow_R$ and $\longrightarrow_S$ *commute weakly* if for all X, Y, and Z such that $X \longrightarrow_R Y$ and $X \longrightarrow_S Z$, there exists W such that $Y \twoheadrightarrow_S W$ and $Z \twoheadrightarrow_R W$.

$\longrightarrow_R$ and $\longrightarrow_S$ *commute* if for all X, Y, and Z such that $X \twoheadrightarrow_R Y$ and $X \twoheadrightarrow_S Z$, there exists W such that $Y \twoheadrightarrow_S W$ and $Z \twoheadrightarrow_R W$.

$\longrightarrow_R$ is *locally confluent* if $\longrightarrow_R$ commutes weakly with itself.

$\longrightarrow_R$ is *confluent* (or *Church-Rosser*) if $\longrightarrow_R$ commutes with itself.

$\longrightarrow_R$ is *noetherian* (or *strongly normalizing*) if there is no infinite sequence of the form

$$X_1 \longrightarrow_R X_2 \longrightarrow_R \cdots \longrightarrow_R X_n \longrightarrow_R \cdots$$

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 *α-conversion*, i.e., we will feel free to say

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

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

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

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

where y and the x_i are arbitrary variables and the P_j are arbitrary λ-terms.

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

$$\lambda x.N$$

or

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

for arbitrary variable x and arbitrary λ-term N and terms P_i.

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

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

σ 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 $\sigma\colon M_0 \twoheadrightarrow_R M_n$.

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

$$\begin{array}{rcl} n \in \mathcal{N} & \Longrightarrow & n \in \mathrm{Ter}(\lambda^{\mathrm{DB}}) \\ M, N \in \mathrm{Ter}(\lambda^{\mathrm{DB}}) & \Longrightarrow & (MN) \in \mathrm{Ter}(\lambda^{\mathrm{DB}}) \\ M \in \mathrm{Ter}(\lambda^{\mathrm{DB}}) & \Longrightarrow & \lambda.M \in \mathrm{Ter}(\lambda^{\mathrm{DB}}) \end{array}$$

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

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

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

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

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

References

[ACCL90] 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, 1990.

[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.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages*, pages 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag. Lecture Notes in Computer Science 259.

[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 namefree 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.

[Fie90] John Field. *Incremental Reduction and its Applications*. PhD thesis, Cornell University, 1990. (Forthcoming).

[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.

[Hue86] G. Huet. Formal structures for computation and deduction (preliminary edition). Unpublished Course Notes, Carnegie-Mellon University, 1986.

[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.

[Kat90] Vinod Kathail. *An Optimal Interpreter for the λ-calculus.* PhD thesis, Massachussetts Institute of Technology, 1990.

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

[Lam90] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, 1990.

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

[Lév75] Jean-Jacques Lévy. An algebraic interpretation of the $\lambda\beta$K-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.

[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.