Incremental Reduction in the Lambda Calculus*

John Field**
Tim Teitelbaum**

TR 90-1130
May 1990

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

*This paper is to be presented at the ACM Conference on Lisp and Functional Programming, Nice, June 1990. This research was supported by NSF and ONR under NSF Grant CCR-8514862, and by ONR under contract number N00014-88-K-0594.
**Authors' Address: Department of Computer Science, Cornell University, Upson Hall, Ithaca, NY 14853. Electronic mail: field@cs.cornell.edu, tt@cs.cornell.edu.

Incremental Reduction in the Lambda Calculus*

John Field   Tim Teitelbaum
Cornell University†

Abstract

An *incremental* algorithm is one that takes advantage of the fact that the function it computes is to be evaluated repeatedly on inputs that differ only slightly from one another, avoiding unnecessary duplication of common computations.

We define here a new notion of incrementality for reduction in the untyped λ-calculus and describe an incremental reduction algorithm, Λ^{inc}. We show that Λ^{inc} has the desirable property of performing *non-overlapping* reductions on related terms, yet is simple enough to allow a practical implementation. The algorithm is based on a novel λ-reduction strategy that may prove useful in a non-incremental setting as well.

Incremental λ-reduction can be used to advantage in any setting where an algorithm is specified in a functional or applicative manner.

1 Introduction

1.1 Incremental Computation

Applications that require computations on a sequence of slightly differing inputs occur with surprising frequency, and researchers have investigated numerous notions of incrementality: Demers, Reps, and Teitelbaum [DRT81,Rep82] proposed algorithms to update functions on syntax trees described by attribute grammars. Alpern, et al. [ACR+87] describe a generalization of incremental attribute grammar evaluation for arbitrary graphs. Earley and Paige and Koenig [Ear76,PK82] have investigated what are in essence incremental methods for updating aggregate data

*This paper is to be presented at the ACM Conference on Lisp and Functional Programming, Nice, June, 1990. This research was supported by NSF and ONR under NSF grant CCR8514862, and by ONR under contract number N00014-88-K-0594.

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

structures (such as sets) in loops in very high-level programming languages. Yellin and Strom [YS87] describe a special programming language where networks of functions on aggregate data structures can be defined. Small changes to such aggregates are reflected in efficient propagation of updates throughout the network. Pugh and Teitelbaum [PT89] discuss incremental evaluation of functions using caching techniques. Berman, Paull, and Ryder [BPR85] have investigated the limitations of incremental algorithms from the abstract point of view of computational complexity. Bengelloun [Ben82] describes an incremental evaluator for the language SCHEME in which list structures manipulated by functions may be incrementally updated. There are countless more specific (or ad-hoc) examples of incremental computation in other domains.

The advantage of investigating incremental reduction in the λ-calculus is that it is more general than the approaches discussed above, yet it is also simple—one can describe changes to functions (including higher-order functions) and to data structures (encoded as λ-terms) in the same formalism. The λ-calculus also has a rich reduction theory that can be exploited to analyze the limits of incremental techniques.

1.2 Prerequisites

The reader is referred to Appendix A for a brief review of λ-calculus terminology we will use here, some of which is slightly nonstandard. An acquaintance with the work on Categorical Combinators of Curien [Cur86a,Cur86b], or the related systems of [ACCL90] and [Fie90b] would be useful. The latter is used extensively in the sequel, and a review of it may be found in Appendix B. The formal treatment of incremental reduction in Section 6 requires some familiarity with the reduction theory of the λ-calculus covered in [Bar84,BKKS87,Klo80,Lév78,Lév80].

2 Motivation

As an example of an application for which incremental λ-reduction is particularly appropriate, consider the abstract syntax for a small expression language in Figure 1, which we will call L.

A program of L consists of a list of declarations and an expression. The expressions of L are built from natural numbers, identifiers, and sums. A declaration is a binding of an identifier to a number. Though patently useless, the language serves to illustrate some important issues. A typical expression of L is the following:

$$\textbf{let}\ a = 2; b = 5\ \textbf{in}\ ((3 + 4) + a)$$

Grammar		*Denotation*	
$\mathit{Prog} ::= \mathit{Decls}\;\mathit{Expr}$	(program)	$[\![\mathit{Prog}]\!]$	$\equiv ([\![\mathit{Expr}]\!][\![\mathit{Decls}]\!])$
$\mathit{Expr} ::= \mathit{Nat}$	(constant)	$[\![\mathit{Expr}]\!]$	$\equiv \lambda\, \mathit{env}\, .\, \mathit{Nat}$
$\mid \mathit{Id}$	(variable)		$\equiv \lambda\, \mathit{env}\, .(\mathit{env}\, \mathit{Id})$
$\mid \mathit{Expr}_1\; \mathit{Expr}_2$	(sum)		$\equiv \lambda\, \mathit{env}\, .(([\![\mathit{Expr}_1]\!]\; \mathit{env}) +([\![\mathit{Expr}_2]\!]\; \mathit{env}))$
$\mathit{Decls} ::= \mathit{Nulldecls}$	(empty decl.)	$[\![\mathit{Decls}]\!]$	$\equiv \lambda\, i\, .(-1)$
$\mid \mathit{Bind}\; \mathit{Decls}$	(decl. list)		$\equiv \lambda\, i\, .\mathbf{if}\, (([\![\mathit{Bind}]\!]\; i) \neq (-1))\ \mathbf{then}\, ([\![\mathit{Bind}]\!]\; i)\ \mathbf{else}\, ([\![\mathit{Decls}]\!]\; i)$
$\mathit{Bind} ::= \mathit{Id}\; \mathit{Nat}$	(binding)	$[\![\mathit{Bind}]\!]$	$\equiv \lambda\, i\, .\mathbf{if}\, (i = \mathit{Id})\ \mathbf{then}\, \mathit{Nat}\ \mathbf{else}\, (-1)$

Figure 1: The Language L

(where we add appropriate keywords for the sake of readability).

Along with each grammar rule in L, we define a *denotation* for the rule's left-hand nonterminal (e.g., $[\![\mathit{Prog}]\!]$), using a notational variant of the lambda calculus in which integer operations, conditionals, etc., are predefined. By reducing the λ-expression denoting a program of L to weak head normal form, we obtain its *meaning*. Thus the meaning of the expression above is 9, obtained by reduction of the λ-expression that denotes it. Most of the complications in the denotational definitions of L arise from the "lookup" mechanism used for determining values to which variables are bound. The abstract syntax of L may be viewed as defining a term algebra of the "parse trees" for the language in the usual way. Editing an expression of L thus amounts to performing a subtree replacement in a term of its algebra, for which there is a corresponding subtree replacement in the λ-expression denoting the term.

Consider the possible consequences of changing part of the program above (by making a subtree replacement in its corresponding term), e.g.,

1. Changing the binding for b to 7.

2. Changing the binding for a to 3.

3. Adding another binding for a.

If we perform (1), there is no need to re-evaluate the sum since it contains no instance of variable b. (2) requires re-evaluating the outer sum to determine its new meaning, but not the inner expression $(3 + 4)$. (3) may invalidate the use of the previous binding for a, as well as require re-evaluation of the outer sum. Multiple changes would have further-reaching consequences.

We would like it to be the case that the new denotational λ-expression can be re-evaluated without repeating any of the work required to evaluate the denotation of the old term—in other words, we want to reduce the new λ-expression *incrementally*. Note that the λ-expression defining a denotation is not subject to completely arbitrary changes after a subtree replacement; only λ-terms denoting corresponding subterms in the language's term algebra are subject to alteration. For instance, in the λ-term denoting a sum,

$$\lambda\, env\,.(([\![Expr_1]\!]\; env) + ([\![Expr_2]\!]\; env))$$

the only λ-subterms that can be replaced are $[\![Expr_1]\!]$ and $[\![Expr_2]\!]$.

3 The Problem

Moving away from our motivating example, let us examine the problem of incremental λ-reduction in the abstract.

3.1 Reduction of Similar Terms

Consider the following λ-terms (where $I \equiv \lambda x.x$):

$$\begin{aligned} M_1 &\equiv (\boxed{\lambda\text{xy.xy}}\,(II)\,(II)) \\ M_2 &\equiv (\boxed{\lambda\text{xy.x}}\,(II)\,(II)) \\ M_3 &\equiv (\boxed{\lambda\text{xy.y}}\,(II)\,(II)) \end{aligned}$$

M_1, M_2, and M_3 are identical except for the boxed terms. The following set of (outermost) reductions suffice to reduce these terms to normal form:

$$\begin{array}{rll} \rho_1: & (\boxed{\lambda\text{xy.xy}}\,(I^aI)\,(I^bI)) & \equiv M_1 \\ \twoheadrightarrow & (I^aI)\,(I^bI) & \\ \xrightarrow{a} & I\,(I^bI) & \\ \longrightarrow & I^bI & \\ \xrightarrow{b} & I & \equiv M_1' \end{array}$$

$$
\begin{array}{llll}
\rho_2: & (\boxed{\lambda \mathrm{xy.x}}\,(I^aI)\,(I^bI)) & \equiv M_2 \\
& \twoheadrightarrow \quad I^aI & \\
& \xrightarrow{a} \quad I & \equiv M_2' \\
\rho_3: & (\boxed{\lambda \mathrm{xy.y}}\,(I^aI)\,(I^bI)) & \equiv M_3 \\
& \twoheadrightarrow \quad I^bI & \\
& \xrightarrow{b} \quad I & \equiv M_3'
\end{array}
$$

Certain redexes above have been labeled by attachment of a superscript letter to their abstraction term. Contraction of such a redex is indicated by the correspondingly labeled '$\longrightarrow$'.

The redexes contracted in ρ_1, ρ_2, and ρ_3 *overlap*, in the sense that ρ_1 and ρ_2 both contract the redex labeled a, and ρ_1 and ρ_3 both contract the redex labeled b.

If we reduce M_1 and subsequently are faced with the task of reducing M_2 and M_3, it would be useful to take advantage of the fact that certain redexes in M_1 have already been contracted, avoiding contracting the same redexes in the reductions of M_2 and M_3.

The similarity among M_1, M_2, and M_3 is made more apparent if we note that they can be represented as substitution instances of the same term:

$$
\begin{aligned}
M_1 &\equiv N[z := \lambda xy.xy] \\
M_2 &\equiv N[z := \lambda xy.x] \\
M_3 &\equiv N[z := \lambda xy.y]
\end{aligned}
$$

where $N \equiv (z\,(II)\,(II))$.

We will refer to such terms as *similar by substitution* (or, in the context of this paper, simply "similar"), since they are the same modulo differing substitutions for some free variable in a term. We will use the notation $M_1 \overset{N}{\sim} M_2$ if M_1 and M_2 can be rewritten as $N[z := P_1]$ and $N[z := P_2]$ for some z, P_1, and P_2. The terms P_1 and P_2 are deemed the *substituands* of the similar terms, and N will be called the *common term* (of M_1 and M_2). A *set of similar terms*, e.g., S^N will refer to a set of pairwise similar terms all having the same common term (e.g., N).

Note that the common term is *not* necessarily a subterm of any of the set of similar terms—the appearance of N in $N[z := P_i]$ is merely a notational convenience. For instance, $(z\,(II)\,(II))$ is not a subterm of M_1. However, the common term of a set of similar terms always can be explicitly "factored out" using β-expansion:

$$
\begin{array}{rcl}
((\lambda z.N)\,\lambda xy.xy) & \longrightarrow_\beta & M_1 \\
((\lambda z.N)\,\lambda xy.x) & \longrightarrow_\beta & M_2 \\
((\lambda z.N)\,\lambda xy.y) & \longrightarrow_\beta & M_3
\end{array}
$$

3.2 Reduction in Common Subterms

Our goal is to reduce a sequence of similar terms in such a way as to avoid the contraction of common redexes, with the aim of enabling *shorter* (thus faster) reductions of later terms in the sequence than would be possible were each term reduced *ab initio*. We defer a more formal definition of what constitutes a "common contraction" or "common computation" to Section 6, appealing here to intuition.

Our approach to the problem is as follows: each member of a set of similar terms is reduced in such a way as to preserve in the *common* term the relevant computations performed in the reduction of the entire term. The idea is illustrated in the following reduction of M_1, where we let $M_1 \equiv N[z := \lambda xy.xy]$ and $N \equiv (z\,(I^a I)\,(I^b I))$:

$$
\begin{array}{lll}
\rho_1'\colon & N[z := \lambda xy.xy] & \equiv M_1 \\
\xrightarrow{a} & (z\,I\,(I^b I))[z := \lambda xy.xy] & \equiv (\lambda xy.xy)\,I\,(I^b I) \\
\xrightarrow{b} & \underbrace{(z\,I\,I)}_{N'}[z := \lambda xy.xy] & \equiv (\lambda xy.xy)\,I\,I \\
\twoheadrightarrow & I\,I & \\
\longrightarrow & I & \equiv M_1'
\end{array}
$$

While ρ_1' is not a outermost reduction, it nonetheless reaches a normal form, and has the pleasing property of producing a very useful intermediate subterm, $N'[z := \lambda xy.xy]$. This term results from the contraction of redexes labeled a and b, which are in a sense relevant only to the common term N. We can summarize ρ_1' as follows:

$$\rho_1'\colon\ N[z := \lambda xy.xy] \twoheadrightarrow N'[z := \lambda xy.xy] \twoheadrightarrow I$$

If we were able to extract the reduced common term N' from its substitution instance M_1 and preserve it in the course of the reduction, we could use it to reduce M_2 and M_3 as follows: Let $N' \equiv (zII)$. We then have:

$$
\begin{array}{lll}
\rho_2'\colon & N'[z := \lambda xy.x] & \equiv \lambda xy.x\,I\,I \\
 & \twoheadrightarrow\ I & \equiv M_2'
\end{array}
$$

$$
\begin{array}{lll}
\rho_3'\colon & N'[z := \lambda xy.y] & \equiv \lambda xy.y\,I\,I \\
 & \twoheadrightarrow\ I & \equiv M_3'
\end{array}
$$

We also know that since $N \twoheadrightarrow N'$, and $M_i \twoheadrightarrow N'[z := P_i]$, ρ_2' and ρ_3' yield correct normal forms for M_2 and M_3, respectively, though their initial terms are not M_2 and M_3.

Unlike their counterparts ρ_2 and ρ_3, ρ_2' and ρ_3' avoid contracting redexes labeled a and b, since these computations were previously carried out in reduction ρ_1', and are embodied in N'. In the sequel, we will demonstrate that the common term and its reduction can indeed be effectively preserved in the course of reducing one of its substitution instances.

Given a set of similar terms, it should be clear that in general it is not possible simply to extract and reduce the common term to a normal form (as we did with N'), as none may exist. Let $R \equiv (z\,(II)\,\Omega)$, where $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. We then have the following outermost reductions:

$$\begin{array}{ll}\tau_1\colon & R[z := \lambda xy.x] \equiv Q_1 \longrightarrow \lambda xy.x\,(II)\,\Omega \longrightarrow II \longrightarrow I \\ \tau_2\colon & R[z := \lambda xy.a] \equiv Q_2 \longrightarrow \lambda xy.a\,(II)\,\Omega \longrightarrow a \\ \tau_3\colon & R[z := \lambda xy.y] \equiv Q_3 \longrightarrow \lambda xy.y\,(II)\,\Omega \longrightarrow \Omega \longrightarrow \cdots\end{array}$$

It should be evident from the example above that the redexes contracted in R depend on R's substituand. In τ_1, $\lambda xy.x$ "caused" redex (II) in R to be contracted. In τ_2, however, the (II) redex is not contracted at all. τ_3 never reaches a normal form (and thus none exists for Q_3). In short, whether or not a redex in common term R is *needed* to yield a normal form depends on the value of the substituand. The notion of "neededness" is formalized and discussed in some detail in [BKKS87].

The possibility of contracting redexes in the common term in advance of the reduction of the entire term is limited by the fact that knowing which redexes in the common term will be subsequently needed is undecidable. At best, the common term can only be reduced to some "safe" form (such as head normal form), but this would not necessarily capture all the contractions performed in the common term during the reduction of one of its substitution instances.

Such safe pre-computation falls under the general rubric of *partial evaluation*, a technique that has been proposed for the implementation of compiler construction systems, among other things (see, e.g., [JSS89]). The goal here is rather different: to preserve computations in the common term that are *dependent* on the substitutable part of an expression *in addition to* those that are independent.[1]

We are thus led to investigate techniques that allow the preservation of the result of contractions in the common term as a *side-effect* of reducing one of the members of a set of similar terms.

3.3 Incremental Lambda-Reduction

In general, we assume we are given terms M_1 and M_2 such that $M_1 \overset{N}{\sim} M_2$, that is, terms having the forms $N[z := P_1]$ and $N[z := P_2]$. We would then like to perform

[1]We have recently become aware of work on incremental computation using partial evaluation [Sun90], but were unable to obtain information on its details in time to permit comparison with the methods used here.

reduction

$$\tau_1\colon N[z := P_1] \longrightarrow\!\!\!\!\!\rightarrow M_1'$$

while also computing a term N' such that $\tau_1^N\colon N \longrightarrow\!\!\!\!\!\rightarrow N'$, where (informally) N' embodies all of the contractions performed in N during τ_1. τ_1^N will be called a *projection* of τ_1 on (common) term N. We can then use N' in a reduction of M_2:

$$\tau_2\colon N'[z := P_2] \longrightarrow\!\!\!\!\!\rightarrow M_2'$$

This process can easily be extended to a sequence of reductions of an arbitrarily large set of similar terms; e.g., from τ_2, we can compute an N'', the final term of a projection $\tau_2^{N'}\colon N' \longrightarrow\!\!\!\!\!\rightarrow N''$ of τ_2 on N', etc.

Note that depending on N', it is generally *not* the case that $N[z := P_2] \longrightarrow\!\!\!\!\!\rightarrow Q$ implies $N'[z := P_2] \longrightarrow\!\!\!\!\!\rightarrow Q$ (although $N[z := P_2] =_\beta N'[z := P_2]$). We will, however, be concerned here with applications in which the desired final term of a reduction is some sort of normal form (e.g., weak head normal form), where *any* final term of the desired form is acceptable. Our goal will be to minimize as much as possible the number of β-contractions required to yield such a normal form by taking advantage of previous computations. The process described above constitutes *incremental reduction* in the λ-calculus.

We have heretofore described only terms that differ in a single subterm. However, the notion of "similarity" can be extended easily to terms that may differ in arbitrarily many subterms (e.g., let $M_1 \equiv N[z_1 := P_{11}\ ,\ z_2 := P_{12}]$ and $M_2 \equiv N[z_1 := P_{21}\ ,\ z_2 := P_{22}]$). We will also consider terms whose substituands may be *nested*, i.e., terms in which substituands contain other substituands. The language denotation discussed in Section 2 has substituands of this form induced by the term structure of its grammar. However, we will defer discussion of these generalizations, (which are of far greater practical value than the case of single substituands), to Section 8. The special case of similar terms with single substituands suffices here for the purposes of exposition.

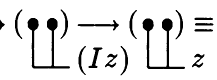
4 Informal Presentation of Incremental Reduction

4.1 Preliminaries

4.1.1 Lambda Terms as Graphs

Our incremental λ-reduction algorithm will use the common device of representing λ-terms by (directed acyclic) *graphs*, an idea pioneered by Wadsworth [Wad71]. In such a representation, subgraphs may be shared, enabling simultaneous contraction of the various redexes represented by the shared term. In addition, implementations will generally allow nodes in the graph to be overwritten in place by terms to which they reduce. For more details on the use of graphs in λ-reduction, see [Pey87]. Lévy [Lév78,Lév80] observed that such shared contractions can be formally described as "parallel" reductions. Such reductions are briefly reviewed in Appendix A.

In the sequel, we will feel free to intermix conventional text and graphs, e.g.,

σ: $(\lambda y.(yy))(Iz) \longrightarrow$  $\equiv zz$

represents a reduction where two (Iz) redexes are shared.

4.1.2 Closures and Environments

Also useful will be the notion of *environment*, used in many practical reduction schemes, e.g., those of [FW87,CCM87]. 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:

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

We will use the notation $[T,\ E]$ to represent a closure consisting of term T and environment E. An environment is denoted thus:

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

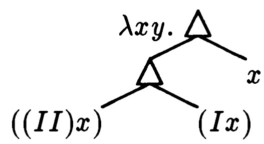
where B_1, B_2, etc. are bindings.

4.1.3 The Fork

Finally, we introduce a new data structure, the *fork* node: Δ. Forks allow certain *sets* of λ-terms to be represented by a single graph. For example, let:

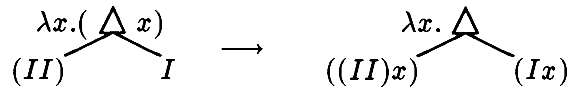
$$Q_1 \equiv \lambda xy.(I\ I\ x)$$
$$Q_2 \equiv \lambda xy.Ix$$
$$Q_3 \equiv \lambda xy.x$$

These terms could be represented simultaneously as follows:



Depending on whether the right or left branches of the two Δ nodes are inspected, different terms may be read off. In the sequel, the two branches will always represent β-equivalent terms (as is the case above).

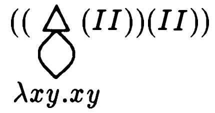
Note that the arguments of an application whose function part is a Δ node can be "distributed" through the Δ to yield an alternate representation of the same set of terms, e.g.:



4.2 Example

Let us return to the example of Section 3 to illustrate how the reduction algorithm Λ^{inc} works, where, as before, we are given the mutually similar terms $M_1 \equiv N[z := \lambda xy.xy]$, $M_2 \equiv N[z := \lambda xy.x]$, and $M_3 \equiv N[z := \lambda xy.y]$.

We begin by reducing M_1 to normal form, using the following graph to represent the initial term:

$M_1 \equiv$ 

Note that the two terms represented by the graph above are the same.

We then reduce M_1 as shown in Figure 2, using a variant of the outermost graph-reduction strategy of Wadsworth, where all steps are either β-contractions (on possibly shared terms) or distribution of arguments of applications through Δ nodes. The node being reduced at each step is boxed. At all times, the graph represents a pair of β-equivalent terms, depending on whether only left branches or only right branches are inspected. The pair of terms represented by the graph at each stage of the reduction process is listed next to each graph.

When reduction of M_1 is complete, we are left with a term headed by a Δ node whose left subtree is the normal form of M_1 and whose right subtree is the term

$$(z\,I\,I)[z := P_1] \equiv N'[z := P_1]$$

If we disregard the substituand P_1, we can regard the right subtree as representing N', the desired final term of the projection of the reduction on N.

In general, Λ^{inc} reduces an initial term whose substituand is replaced by a Δ node (the branches of which both point to the substituand). When a Δ node is encountered in the function part of an application during reduction, its argument is distributed into each branch of the Δ, effectively creating two copies of the application. We can view the Δ node as a sort of "checkpoint" in the reduction process, which preserves the redex whose contraction depends on the value of the substituand (i.e., a redex whose function part is the substituand). Reduction then proceeds only on the *left* branch of the Δ. However, contraction of a redex in the left subterm of a Δ may have the beneficent side-effect of contracting a redex in the right subterm, if the redex is shared. The trick is to perform the reduction in such a way that the maximal number of such shared contractions take place.

The combination of graph reduction and appropriate use of the Δ node is crucial to performing incremental reduction: The Δ node enables a subterm whose reduction is *dependent* upon the substituand to be preserved, while graph reduction enables redex contractions that are *independent* of the substituand to affect both terms reachable from a Δ.

Continuing with the example above, if we remove the topmost Δ node, discard its left subterm, and replace P_1 by

in the (former) right subterm of the topmost Δ, we form the new term (disregarding the Δ node)

$$N'[z := P_2]$$

We then reduce the new term, taking advantage of the fact that N' embodies the reduction already performed in N above, and avoiding (as was our aim) contraction of overlapping redexes. The new reduction is given in Figure 3. We can repeat the process as required for additional substituands P_3, P_4.... Note that in the example above, the final graph is the same as that of the reduction in Figure 2. In general, however, each substituand may induce additional reductions in the common term (which was not possible above since the common term $N' \equiv (z\,I\,I)$ was already reduced to normal form by the first reduction).

Each time reduction is to be performed in the presence of a new substituand, the initial term is reconstructed by replacing each reference to a fork node with a reference to its *right* subterm, thus preserving the "checkpoints" on which the substituand depended. A new substituand (along with an appropriately configured Δ node) replaces the old one and the reduction process begins anew.

Unfortunately, Wadsworth-style graph reduction is not completely satisfactory in the case illustrated in Figure 4. There we are unable to take advantage of the fact that the redex Ia is essentially a copy (i.e., a residual) of the Iy redex in $\lambda y.Iy$. The Iy redex is left uncontracted in the final term. This phenomenon is related to the issue of "full-laziness" in functional language implementations. Although we could have contracted the Iy redex *before* applying $\lambda y.Iy$ to a, we wish to retain an outermost reduction strategy. By using shared closures and environments, the (Iy) redex can be contracted independently of the substitution of a for y, even though the reduction is nominally outermost, as we see in Figure 5.

In order to preserve residuals of inner redexes, Λ^{inc} will use shared environments and closures, as well as shared λ-terms. In addition, by using environments, substitution can be implemented efficiently.

5 The Incremental Reduction Algorithm

5.1 $\Delta\Lambda$CCL

Λ^{inc} is specified using an augmented version of the **ΛCCL** formalism described in [Fie90b]. The system defined in [ACCL90] is quite similar, and would also suffice for this purpose. **ΛCCL** is intended to formalize the notions of *environment* and *closure* used to implement the operation of substitution in many λ-reduction algorithms. **ΛCCL** is a *conservative extension* of the $\lambda\beta$-calculus, that is, for every β-reduction, there is a corresponding reduction in **ΛCCL**. This close connection between **ΛCCL** and the λ-calculus makes it easy to show that our algorithm is correct.

By manipulating environments and closures explicitly, **ΛCCL** makes possible λ-reduction strategies in which certain substitution operations may be deferred until the need for a term arises, or in which a term may be reduced independently of several different substitutions for its free variables. Furthermore, as a left-linear term rewriting system, **ΛCCL** reduction can be implemented in a straightforward way using graph-reduction on shared terms (see e.g., [BvEG+87] for formalizations of this idea). All of these properties of **ΛCCL** will be exploited in Λ^{inc}. The details of **ΛCCL** are reviewed in Appendix B.

To implement *incremental* reduction, we will extend **ΛCCL** with a new constructor, $\Delta(\cdot, \cdot)$, the "fork" node used above informally, and add several new axioms defining its behavior. The resulting system is called **ΔΛCCL**:

Definition 5.1 *The terms of* **ΔΛCCL** *are those of* **ΛCCL**, *along with those built from the following new constructor:*

$$\Delta(\mathcal{L}, \mathcal{L}) : \mathcal{L} \quad \textit{(fork)}$$

The axioms of **ΔΛCCL** *are those of* **ΛCCL**, *augmented with the following:*

$$\begin{array}{ll} (\Delta\mathbf{C}) & [\Delta(A, B), E] = \Delta([A, E], [B, E]) \\ (\Delta\Lambda) & \Lambda(\Delta(A, B)) = \Delta(\Lambda(A), \Lambda(B)) \\ (\Delta\mathbf{Apply}) & \mathrm{Apply}(\Delta(A, B), C) = \Delta(\mathrm{Apply}(A, C), \mathrm{Apply}(B, C)) \end{array}$$

These new axioms formalize the distribution rules for Δ used earlier.

Treated as a term-rewriting system, **ΔΛCCL** is well behaved, i.e., we have the following:

Proposition 5.1 **ΔΛCCL** *is confluent.*

We can interpret a **ΔΛCCL** term as a *set* of related **ΛCCL** terms (or λ-terms). A particular term of the set is determined by specifying a path to be taken through the Δ nodes in the term (i.e., for each Δ, whether to inspect the right or the left subterm). Two particularly useful terms to consider are those designated by inspecting only left subterms of Δ nodes, or only right subterms:

Definition 5.2 *Let T be a* **ΔΛCCL** *term. Then LeftInterp(T) is defined as follows:*

$$\begin{array}{rcl} \textit{LeftInterp}(\Delta(A, B)) & = & \textit{LeftInterp}(A) \\ \textit{LeftInterp}(\mathrm{Apply}(A, B)) & = & \mathrm{Apply}(\textit{LeftInterp}(A), \textit{LeftInterp}(B)) \\ \textit{LeftInterp}(\Lambda(A)) & = & \Lambda(\textit{LeftInterp}(A)) \\ \textit{LeftInterp}(\mathrm{Var}) & = & \mathrm{Var} \\ & \vdots & \end{array}$$

(similarly for other **ΔΛCCL** *operators)*

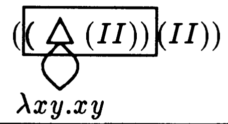
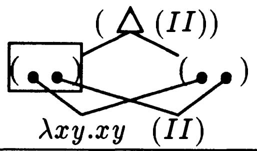
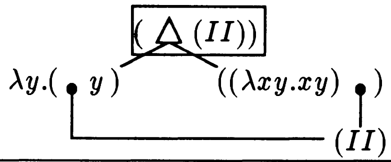
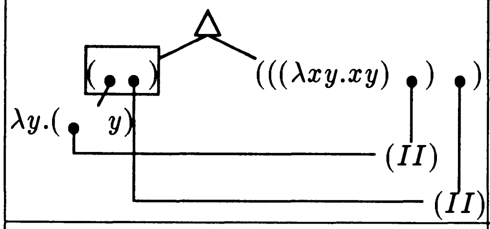
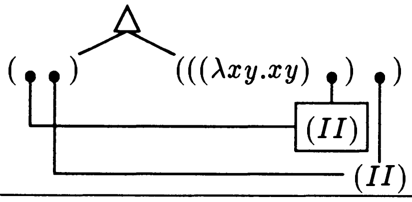
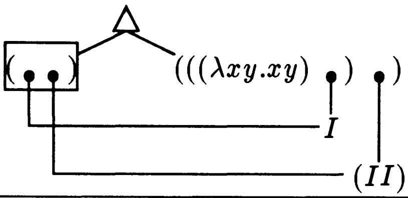
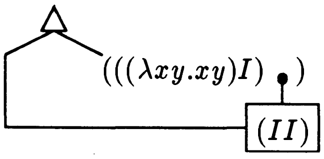
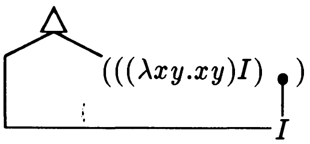
Graph	Op.	Corresponding Terms	
		Left Δ-Subterms	Right Δ-Subterms
	dist.	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda xy.xy)(II)(II))$	$((\lambda xy.xy)(II)(II))$
	dist.	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((\lambda y.(II)\,y)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$((II)(II))$	$((\lambda xy.xy)(II)(II))$
	β	$(I\,(II))$	$((\lambda xy.xy)\,I\,(II))$
	β	(II)	$((\lambda xy.xy)\,I\,(II))$
	β	I	$((\lambda xy.xy)\,I\,I)$

Figure 2: Reduction of M_1

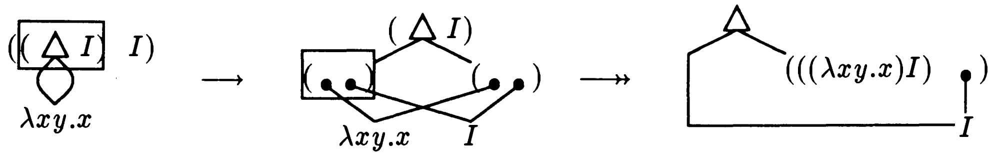


Figure 3: Contraction of $N'[z := P_2]$

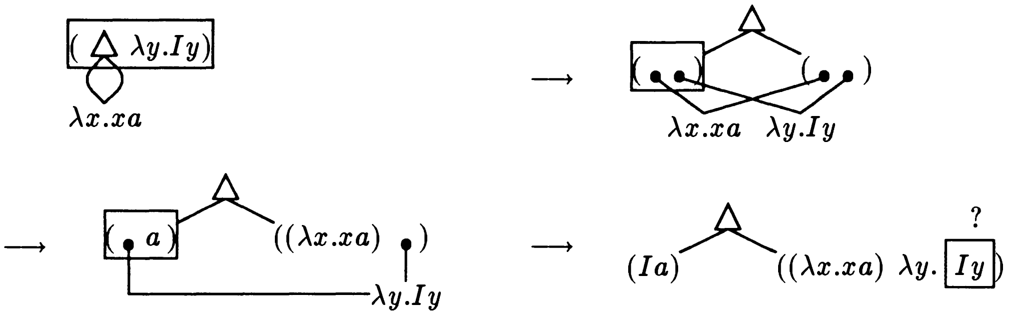


Figure 4: Shortcomings of Outermost Graph Reduction

RightInterp(T) is defined in the analogous manner for right subterms of Δs.

Given an arbitrary $\mathbf{\Delta\Lambda CCL}$ reduction, we can construct its image on the $\mathbf{\Lambda CCL}$ terms formed by following only left or only right branches of Δs:

Proposition 5.2 *Given* $\mathbf{\Delta\Lambda CCL}$ *reduction*

$$\rho \colon A \twoheadrightarrow_{\Delta\Lambda\mathrm{CCL}} B$$

there exists an interpretation of ρ*, LeftInterp*(ρ)*, such that*

$$\mathit{LeftInterp}(\rho) \colon \mathit{LeftInterp}(A) \twoheadrightarrow_{\Lambda\mathrm{CCL}} \mathit{LeftInterp}(B)$$

where the number of **(Beta)** *contractions in LeftInterp*(ρ) *is less than or equal to the number of* **(Beta)** *contractions in* ρ.

There also exists a symmetric interpretation, *RightInterp*(ρ).

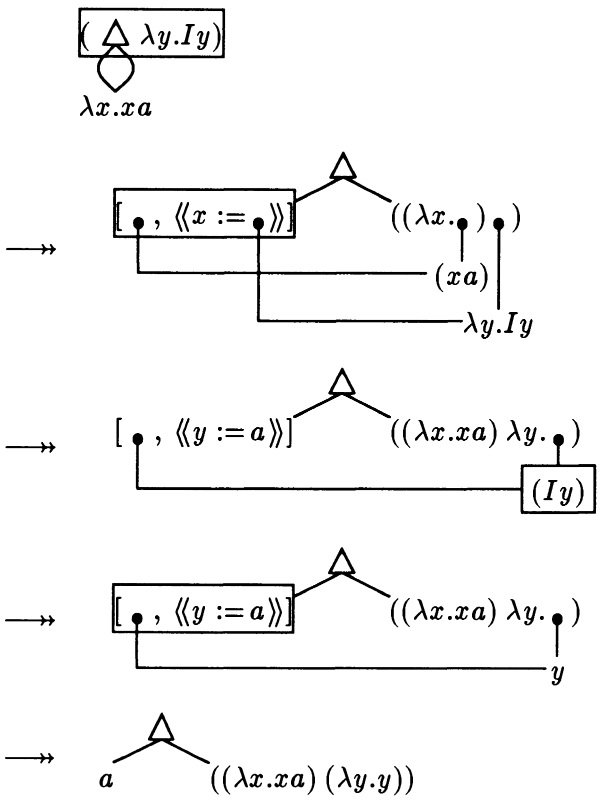


Figure 5: Reduction using Closures and Environments

5.2 Reduction Strategy

Figure 6 consists of two versions of an algorithm that reduces a term of **ΔΛCCL** to the **ΛCCL**-equivalent of weak head normal form, *WHNF*.

The first algorithm, *rwhnf*() is the starting point for Λ^{inc}. It is obtained by omitting the lines preceded by '*'s in Figure 6, and operates only on terms of **ΛCCL**. *rwhnf*() is *weak-head normalizing*, and reduces only *needed* redexes (see Section 6.1). Its reduction strategy nevertheless differs from that normally used by environment-based λ-reduction algorithms, e.g., that of [HM76] (see below).

Δ*rwhnf*() (the full algorithm in Figure 6) is an extension of *rwhnf*() to terms of **ΔΛCCL**. Most of the properties of Δ*rwhnf*() are straightforward consequences of the properties of *rwhnf*(). However, by inserting Δ nodes at appropriate points in the initial term, Δ*rwhnf*() will be used to perform incremental λ-reduction.

rwhnf() and Δ*rwhnf*() are specified using the rewrite rules of **$\Delta\Lambda$CCL**, 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 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 **$\Delta\Lambda$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, while retaining the original references (pointers) to A's *sub*terms, if any. *rpenf*() reduces an environment to a simple normal form, *PENF*.

5.2.1 Behavior of *rwhnf*()

Let us examine some of the salient features of *rwhnf*(). Our goal is to ensure that no redex shared between different subterms of a Δ is ever copied, thus ensuring that reduction of a redex in a left Δ subterm is always shared with its counterpart in the right subterm, if any exists. This property is crucial for ensuring that Λ^{inc} does indeed perform non-overlapping reductions

A **ΛCCL** term not in *whnf* (thus a candidate for reduction by *rwhnf*()) must have an outer constructor that is either an application or a closure. In the former case, *rwhnf*() reduces the function part of the application to *whnf* (in line **L1** of the algorithm), simulating the usual outermost strategy. More unusual is the latter case (line **L2**), in which the term part of a closure is reduced before the closure itself is reduced. This is done to prevent reduction of the closure from causing a redex in its term part to be copied.

In line **L3**, we see that an environment must be "pushed inside" an abstraction before the abstraction is considered to be in *whnf*. Most other reduction schemes that use environments eschew this step, regarding a term of the form $[\Lambda(A),\ E]$ as a *whnf*. This requires an alternate (**Beta$'$**) axiom that operates on an application whose function part is a closure (see, e.g., [ACCL90]). Once again, the rationale for our alternate strategy is to maximize sharing.

In Line **L4**, we see that a term in an environment is reduced to *whnf* before its top constructor is copied by *CopyTop*(), which could otherwise copy a shared redexes.

Taken together, the properties of *rwhnf*() outlined above make it particularly suitable as the basis for incremental reduction.

5.3 Plan of the Algorithm

The general plan of the algorithm is as follows: Given a sequence of similar λ-terms[2]

$$\begin{array}{rcl} M_1 & \equiv & N[z := P_1] \\ M_2 & \equiv & N[z := P_2] \\ & \vdots & \end{array}$$

Λ^{inc} effectively computes

$$M_1', M_2', \ldots \in whnf$$

as well as

$$N_1[z := P_1'], N_2[z := P_2'], \ldots$$

such that for all i,

$$\begin{array}{rcl} M_i & \twoheadrightarrow_\beta & M_i' \\ P_i & \twoheadrightarrow_\beta & P_i' \\ N_{i-1} & \twoheadrightarrow_\beta & N_i \end{array}$$

(where $N_0 \equiv N$). For each i, rather than reduce M_i directly, the algorithm reduces $N_{i-1}[z := P_i]$ to M_i' (in the process yielding $N_i[z := P_i']$ as well).

Note that while $N[z := P_1]$ and M_1 refer to the same λ-term, the substitution notation used in the former case has an *explicit* counterpart in the **ΛCCL** term

$$[[\![N]\!]_{\Lambda\mathrm{CCL}}, \langle \emptyset, [\![P_1]\!]_{\Lambda\mathrm{CCL}} \rangle]$$

which differs from $[\![M_1]\!]_{\Lambda\mathrm{CCL}}$, although the two are interconvertible. In Λ^{inc}, we will use the former as the initial term on which the algorithm operates to ensure that P_1 is represented by a single node in the graph.

[2]We assume without loss of generality that z is the only free variable in N.

5.4 Λ^{inc}

The algorithm Λ^{inc} is given in Figure 7. We assume we are given the **ΛCCL** translation of the common term, $[\![N]\!]_{\Lambda\mathrm{CCL}}$, after which the algorithm iteratively processes a sequence of substituands translated into **ΔΛCCL**:

$$[\![P_1]\!]_{\Lambda\mathrm{CCL}}, [\![P_2]\!]_{\Lambda\mathrm{CCL}}, \ldots$$

At each iteration, the algorithm constructs a **ΔΛCCL** term A_i, and applies $\Delta rwhnf()$ to A_i to yield a new term A_i'.

Note that Step (2) of Λ^{inc} depends on the fact that there is only a single instance of $[\![P_{i-1}]\!]_{\Lambda\mathrm{CCL}}$ (perhaps with multiple references) in $\hat{A}_{i-1}$, and that each phase of the algorithm causes

$$\Delta([\![P_i]\!]_{\Lambda\mathrm{CCL}}, [\![P_i]\!]_{\Lambda\mathrm{CCL}})$$

to be overwritten by $[\![P_{i-1}]\!]_{\Lambda\mathrm{CCL}}$.

Note also that in the examples considered thus far, the reduced term has only one Δ node. In general, however, multiple Δ nodes may appear in the final term as a result of self-application.

We now claim that Λ^{inc} is *correct*, that is:

Proposition 5.3 *For all $i = 1\ldots,$ the terms A_i and A_i' constructed by Λ^{inc} have the following properties:*

$$\begin{aligned}
[\![\mathit{LeftInterp}(A_i)]\!]_\lambda &= [\![\mathit{RightInterp}(A_i)]\!]_\lambda \\
&= N_{i-1}[z := P_i] \\
[\![\mathit{LeftInterp}(A_i')]\!]_\lambda &= M_i' \in whnf \\
[\![\mathit{RightInterp}(A_i')]\!]_\lambda &= N_i[z := P_i']
\end{aligned}$$

and

$$\begin{aligned}
M_i &\twoheadrightarrow_\beta M_i' \\
P_i &\twoheadrightarrow_\beta P_i' \\
N_{i-1} &\twoheadrightarrow_\beta N_i
\end{aligned}$$

(where $N_0 \equiv N$)

Proposition 5.3 follows from the properties of **ΔΛCCL** discussed in Section 5.1, as well as the following fact: Every contraction performed by $\Delta rwhnf()$ preserves the invariant

$$[\![\mathit{RightInterp}(B)]\!]_\lambda = \hat{N}[z := P_i]$$

```
Δrwhnf(T) ≡
case T of
*     Δ(T'_1, T'_2): rwhnf(T'_1);
      Λ(A): skip;                                             {T ∈ WHNF}
      Var: skip;                                              {T ∈ WHNF}
      Apply(A, B): seq
{L1}      rwhnf(A);
          case A of
*             Δ(A'_1, A'_2): seq
*                 T := Δ(Apply(A'_1, B):T̂, Apply(A'_2, B));    {rule (ΔApply)}
*                 rwhnf(T̂)
*                 endseq
              Λ(A'): seq
                  T := [A', ⟨∅, B⟩];                          {rule (Beta)}
                  rwhnf(T)
                  endseq;
              else: skip                                      {T ∈ WHNF}
          endseq;
      [L, E]: seq
{L2}      rwhnf(L);
          case L of
*             Δ(L'_1, L'_2): seq
*                 T := Δ([L'_1, E]:T̂, [L'_2, E]);             {rule (ΔC)}
*                 rwhnf(T̂)
*                 endseq;
              Apply(A, B): seq
                  T := Apply([A, E], [B, E]);                 {rule (DApply)}
                  rwhnf(T)
                  endseq;
{L3}          Λ(A): T := Λ([A, ⟨E ∘ □, Var⟩]);                {rule (DΛ); T ∈ WHNF}
              [L_1, E_1]: seq
                  T := [L_1, E_1 ∘ E];                        {rule (AssC)}
                  rwhnf(T)
                  endseq;
              Var: seq
                  rpenf(E);
                  case E of
                      ∅: T := Var;                            {rule (NullC)}
                      □^n: skip;                              {T ≡ n!; T ∈ WHNF}
                      ⟨E, A⟩: seq
{L4}                      rwhnf(A);
                          T := CopyTop(A)                     {rule (VarRef)}
                      endseq
              endseq
      endseq
```

Figure 6: Algorithms $\Delta rwhnf()$ and $rwhnf()$ (without '*'s)

where $N \twoheadrightarrow_\beta \widehat{N}$ and B is the the root of the graph on which *rwhnf*() operates. We can also show that the substituand is always represented by a single term in its graph (i.e., is never copied in the course of the reduction), and thus that Step (2) of the algorithm is well-defined.

However, we also want to show that Λ^{inc} is indeed *incremental*, in that it avoids contracting redexes unnecessarily at each step, relative to the reductions it has performed earlier. To do this requires an excursion into the reduction theory of the λ-calculus, which we undertake below.

6 Theoretical Machinery

In order to give a rigorous treatment of incremental reduction, we will need to refer to the reduction theory of the λ-calculus developed in Barendregt [Bar84], Barendregt, et al. [BKKS87], Klop [Klo80], and particularly, in Lévy [Lév78,Lév80], with which we must assume the reader is familiar, as space precludes a complete review.

6.1 Preliminaries

The discussion to follow will depend on the following notions from one of the above sources:

— The classical notion of the set of *residuals* of a term R in M relative to a reduction ρ: $M \twoheadrightarrow M'$, notated R/ρ (described in [Bar84]).

— Lévy's notion of *reduction residual*, notated σ/ρ, along with the related concepts of *reduction prefix*, notated $\rho \leq \sigma$, and "strong" equivalence of reduction (or equivalence modulo permutation of redexes), notated $\rho \simeq \sigma$ (described in [Lév78,Lév80,Klo80]).

— Lévy's idea of *family class* and *relative reduction* (discussed in [Lév80]).

— The concept of *neededness* and *needed reduction*, (described in [Lév78,Lév80, BKKS87]).

In the sequel, both λ-reductions and **ΔΛCCL** reductions are assumed (in general) to be *parallel*—i.e., those in which multiple redexes in a given term may be contracted in a single step via shared redexes. (See Appendix A).

6.2 New Definitions

6.2.1 Relative Prefixes of Reductions

The following strengthening of Lévy's notion of reduction prefix will be used:

Definition 6.1 *Let ρ and σ be coinitial reductions. Then ρ is a* relative prefix *of σ, notation $\rho \leq_R \sigma$ if:*

1. $\rho \leq \sigma$
2. *ρ is relative to σ.*

Definition 6.2 *Let ρ and σ be coinitial reductions. Then ρ is* relatively equivalent *to σ, notation $\rho \simeq_R \sigma$ if $\rho \leq_R \sigma$ and $\sigma \leq_R \rho$.*

Proposition 6.1 *$\simeq_R$ is an equivalence relation.*

6.2.2 Overlapping Reductions

Definition 6.3 *Let ρ and σ be coinitial. Then ρ and σ* overlap, *notation $\rho \asymp \sigma$, if there exists a non-null common relative prefix τ and reductions δ_1 and δ_2 such that $\tau + \delta_1 \simeq_R \rho$ and $\tau + \delta_2 \simeq_R \sigma$.*

Proposition 6.2 *Let $R(M)$ be the set of reductions beginning with term M. Let $R(M)/\simeq_R$ be the set of equivalence classes of $R(M)$ defined by the relation $\simeq_R$, whose members are ordered by:*

$$\sigma \leq_R \rho \implies [\sigma] \leq_R [\rho]$$

(where the meaning of $\leq_R$ is here overloaded). Then for all reductions ρ: $M_1 \twoheadrightarrow M_1'$ and σ: $M_2 \twoheadrightarrow M_2'$, there exists a greatest lower bound $[\rho] \sqcap [\sigma]$ in $R(M)/\simeq_R$.

Thus we see that for any pair of overlapping reductions, there exists a "maximal overlap" (modulo $\simeq_R$). Lévy [Lév78] shows that such a greatest lower bound *does not* in general exist for reductions ordered by $\leq$.

6.2.3 Projection of Reduction

We are now in a position to give a more formal definition of projection.

Definition 6.4 *Let $M \equiv C[N]$ (i.e., a term containing subterm N), and let ρ be as follows:*

$$\rho\colon\ M \longrightarrow\!\!\!\!\!\rightarrow M'$$

Then ρ^N is a projection *of ρ on N if:*

$$[\rho^N] = \sqcap\{\sigma \mid \sigma\colon N \longrightarrow\!\!\!\!\!\rightarrow N',\ \textit{and for all}\ \tau \in R(N'),\ \tau \not\preceq (\rho/\sigma)\}$$

The idea of this definition is that the projection of

$$\rho\colon\ M \longrightarrow\!\!\!\!\!\rightarrow M'$$

on N should be the least reduction entirely in N which is needed to compute M' or some term to which M' reduces.

We can construct the projection of a reduction on a given term using a method similar to the Klop's standardization construction [Klo80].

We can extend the definition above in the obvious way to λ-terms of the form $N[z := P]$:

Definition 6.5 *Let $M \equiv N[z := P]$ and $\rho\colon\ M \longrightarrow\!\!\!\!\!\rightarrow M'$. Let σ be the following reduction:*

$$\sigma\colon((\lambda z.N)P) \longrightarrow M$$

Then the projection ρ^N is defined by:

$$\rho^N \equiv (\sigma + \rho)^N$$

6.2.4 Overlapping Reductions on Similar Terms

We can now define overlapping reductions on similar terms:

Definition 6.6 *Let $M_1 \overset{N}{\sim} M_2$ be similar terms; i.e., let $M_1 \equiv N[z := P_1]$ and $M_2 \equiv N[z := P_2]$. Then*

$$\rho_1\colon\ M_1 \longrightarrow\!\!\!\!\!\rightarrow M_1'$$

and

$$\rho_2\colon\ M_2 \longrightarrow\!\!\!\!\!\rightarrow M_2'$$

overlap modulo N*, notation $M_1 \overset{N}{\asymp} M_2$, if*

$$\rho_1^N \asymp \rho_2^N$$

Definition 6.6 gives us the formal counterpart of our requirement that incremental reduction not "repeat any work"— such reductions must be non-overlapping.

6.2.5 Overlapping Non-Coinitial Reductions

Finally, we will need a notion of overlapping for certain non-coinitial reductions:

Definition 6.7 *Let ρ_1 and ρ_2 be reductions of terms $M_1 \equiv N_1[z := P_1]$ and $M_2 \equiv N_2[z := P_2]$, and γ_1 and γ_2 be reductions of N such that:*

$$\begin{array}{lll}\rho_1\colon & N_1[z := P_1] & \longrightarrow\!\!\!\!\to & M_1' \\ \rho_2\colon & N_2[z := P_2] & \longrightarrow\!\!\!\!\to & M_2' \\ \gamma_1\colon & N & \longrightarrow\!\!\!\!\to & N_1 \\ \gamma_2\colon & N & \longrightarrow\!\!\!\!\to & N_2\end{array}$$

Let τ be any reduction in the equivalence class of $\gamma_1 \sqcap \gamma_2$ (which exists by Proposition 6.2) such that

$$\begin{array}{l}\gamma_1 \simeq_R (\tau + \delta_1) \\ \gamma_2 \simeq_R (\tau + \delta_2)\end{array}$$

Then ρ_1 and ρ_2 overlap modulo (N_1, N_2), *notation*

$$\rho_1 \overset{N_1,N_2}{\asymp} \rho_2$$

if

$$(\delta_1 + {\rho_1}^{N_1}) \asymp (\delta_2 + {\rho_2}^{N_2})$$

Given Definition 6.7, the following is immediate:

Proposition 6.3 *Given terms $M_1 \equiv N[z := P_1]$, $M_2 \equiv N[z := P_2]$, N' and reductions*

$$\begin{array}{lll}\rho_1\colon & M_1 & \longrightarrow\!\!\!\!\to & M_1' \\ \rho_2\colon & N'[z := P_2] & \longrightarrow\!\!\!\!\to & M_2'\end{array}$$

such that $\rho_1^N\colon N \longrightarrow\!\!\!\!\to N'$, then

$$\rho_1 \overset{N,N'}{\not\asymp} \rho_2$$

Note that

$$(\rho_1^N + \rho_2)\colon M_2 \longrightarrow\!\!\!\!\to N'[z := P_2] \longrightarrow\!\!\!\!\to M_2'$$

Proposition 6.3 states that one can effectively reduce similar terms with non-overlapping reductions by using the final term of the projection of the first reduction on the common term as the starting point for the second reduction.

7 Properties of Λ^{inc}

We now return to Λ^{inc} with the intention of showing that the reductions used to compute each term are formally non-overlapping:

At each iteration, Λ^{inc} applies $\Delta rwhnf()$ to A_i, yielding A_i'. Let ρ_i be the $\mathbf{\Delta\Lambda CCL}$ reduction performed by Λ^{inc}, such that

$$\rho_i\colon A_i \twoheadrightarrow_{\Delta\Lambda\text{CCL}} A_i'$$

Define the β-reductions corresponding to ρ_i as follows:

$$\begin{aligned}\sigma_i &= \beta(\mathit{LeftInterp}(\rho_i))\\ \tau_i &= \beta(\mathit{RightInterp}(\rho_i))\end{aligned}$$

By Proposition 5.3, we have

$$\begin{array}{lllll}\sigma_i\colon & N_{i-1}[z := P_i] & \twoheadrightarrow_\beta & M_i' \\ \tau_i\colon & N_{i-1}[z := P_i] & \twoheadrightarrow_\beta & N_i[z := P_i']\end{array}$$

We can now show that Λ^{inc} lives up to its name:

Proposition 7.1 *Let ρ_i, σ_i, and τ_i be as defined above. Then*

$$\begin{array}{rlll}\sigma_i^{N_{i-1}}\colon & N_{i-1} & \twoheadrightarrow_\beta & N_i \\ \sigma_i^{P_i}\colon & P_i & \twoheadrightarrow_\beta & P_i'\end{array}$$

and

$$(\sigma_i^{N_{i-1}} + \sigma_i^{P_i}) \quad \simeq_R \quad \tau_i$$

The proof of Proposition 7.1 requires showing that any contraction of a redex R in $[\![\mathit{LeftInterp}(B)]\!]_\lambda$ by $\Delta rwhnf()$ also has the effect of contracting *every* redex in $[\![\mathit{RightInterp}(B)]\!]_\lambda$ of which R is a residual. This property is a consequence of the sharing properties of the reduction strategy used by $rwhnf()$ outlined in 5.2.1, and can be shown formally through the use of a *labeled* version of $\mathbf{\Lambda CCL}$, which is described in [Fie90b].

We see then that Λ^{inc} simultaneously computes a weak head normal form of its initial term, and the projection of the reduction thus performed on the common term of the initial term (as well as he projection on the substituand). By Proposition 6.3, we can conclude that the σ_i are pairwise non-overlapping, and thus that Λ^{inc} is incremental in a well-defined sense.

8 Multiple Substituands

The discussion above has been concerned with incremental reduction of terms differing in a single substituand. However, as the example of Section 2 illustrates, it is much more useful to consider the possibility of multiple, and perhaps nested, substituands. It turns out that Λ^{inc} can be extended almost trivially to deal with this new model. Here we will not attempt to cover all the formal details required to treat the extension, which are given in [Fie90a], and instead provide an overview of the concepts involved.

We will now assume that we are given a λ-term M such that

$$M \equiv N[z_1 := M_1, \ldots, z_n := M_n],\ n \geq 0$$

where each M_i has the form

$$M_i \equiv N_i[z_{i1} := M_{i1}, \ldots, z_{in_i}, := M_{in_i}],\ n_i \geq 0$$

and each of the M_{ij} are assumed have a similar form, etc.

After reducing M to $M' \in whnf$, we would like to replace some set of disjoint substituands

$$M_{\vec{v}_1}, \ldots, M_{\vec{v}_m}$$

in M, and reduce the resulting new term.

Consider, for example, the term $M \equiv (\lambda y.yy\ (I(Ia)))$, in which we designate two substituands z_1 and z_2 as follows:

$$\begin{aligned} M &\equiv (z_1\ (I\, z_2))[z_1 := \lambda y.yy, z_2 := (Ia)] \\ &\equiv N[z_1 := P_1, z_2 := P_2] \end{aligned}$$

To reduce this term incrementally, we start by replacing the substituands z_1 and z_2 by terms with Δ nodes, as before. However, we now *index* the Δ nodes by a number corresponding to the index of the substituands, as shown by the term A in Figure 8.

Let A be the indexed **ΔΛCCL** term representing M i.e., where

$$[\![LeftInterp(A)]\!]_\lambda = [\![RightInterp(A)]\!]_\lambda = M$$

We then proceed to reduce it using $\Delta rwhnf()$ as before. $\Delta rwhnf()$ is oblivious to the values of indices used for Δ nodes, although rules (**ΔApply**) and (**ΔC**) are expected to preserve the values of indices of Δ operators on each side of their respective equations. $\Delta rwhnf()$ performs the reduction $\rho \colon A \twoheadrightarrow A'$, summarized in Figure 8

$\Lambda^{\text{inc}} \equiv$

let $A_0 = [[\![N]\!]_{\Lambda\text{CCL}}, \langle \emptyset, [\![P_1]\!]_{\Lambda\text{CCL}} \rangle]$,
$\quad P'_0 = P_1$
in
for $i = 1 \ldots$ **do**

1) *Recursively traverse* A_{i-1}*, replacing all references to* Δ *nodes with references to the* Δ*'s right child, yielding* $\hat{A}_{i-1}$*;*
2) *Overwrite* $[\![P'_{i-1}]\!]_{\Lambda\text{CCL}}$ *in* $\hat{A}_{i-1}$ *with* $\Delta([\![P_i]\!]_{\Lambda\text{CCL}}, [\![P_i]\!]_{\Lambda\text{CCL}})$*, yielding* A_i*;*
3) *Apply* Δrwhnf() *to* A_i*, yielding* A'_i

Figure 7: Algorithm Λ^{inc}

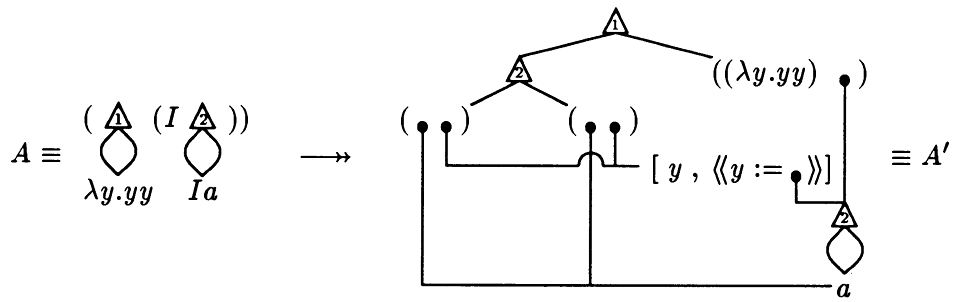


Figure 8: Reduction with indexed Δ nodes

(using our informal notation for environments and closures). Though the final term of the reduction, A', is a bit complicated, its interpretation is straightforward:

$$\begin{aligned}
[\![\mathit{LeftInterp}(A')]\!]_\lambda &\equiv (aa) \\
[\![\mathit{RightInterp}(A')]\!]_\lambda &\equiv (z_1\, z_2)[z_1 := \lambda y.yy, z_2 := a] \\
&\equiv N'[z_1 := P_1', z_2 := P_2']
\end{aligned}$$

Note that the reduction above preserves the contraction of the $(I z_1)$ redex in N and the (Ia) redex in P_2 (there are no redexes in P_1 to preserve).

The indices in Δ nodes only come into play when a subterm is replaced and reduction is to be performed anew. The only alterations required for algorithm Λ^{inc} to behave correctly on multiple substituands (other than changing the initializations appropriately) are these:

- Step (1) of algorithm Λ^{inc} removes all Δ nodes having the same index as the substituands being replaced.

- In Step (2), more than one substituand may be replaced by appropriately configured and indexed Δ nodes.

We can now interpret terms of $\mathbf{\Delta\Lambda CCL}$ by specifying for each index found in a Δ node whether its right or left subterm is to be examined; i.e., instead of *LeftInterp*(A) or *RightInterp*(A), we have indexed interpretations of the form $\mathit{Interp}_S(A)$:

8.1 Indexed Interpretations

Let *ForkIndices*(A) be the set of indices of Δ nodes found in term $A \in \mathbf{\Delta\Lambda CCL}$. Let S be a subset of *ForkIndices*(A). Then (informally) $\mathit{Interp}_S(A)$ is the $\mathbf{\Lambda CCL}$ term gotten by following right branches of Δ nodes with indices $i \in S$, and left branches otherwise.

8.2 Overlapping Contexts

Consider the term

$$\begin{aligned}
M &\equiv (z_1\, ((\lambda x.xa)\, z_2))[z_1 := I, z_2 := I] \\
&\equiv N[z_1 := P_1, z_2 := P_2]
\end{aligned}$$

where we reduce M by $\rho\colon M \twoheadrightarrow a$. In this case, not only do we want Λ^{inc} to compute the projection of ρ on N, P_1, and P_2 as before, but since we are now allowed to replace arbitrary combinations of substituands, we need to compute the projection of ρ on *every* context that could contain some subset of the substituands. For example, if we replace only substituand P_1, we need to ensure that we compute the projection of ρ on Q, where we write M as

$$M \equiv (z_1\,((\lambda x.xa)\,I))[z_1 := I] \equiv Q[z_1 := P_1]$$

Note that the projection ρ^Q is not the concatenation of projections ρ^N and ρ^{P_2}, since together, N and P_2 create a redex, (Ia), that does not exist separately in either term.

It turns out that by choosing the indexed interpretation of a term reduced by $\Delta rwhnf()$ according to the indices of the substituands being replaced, we can reconstruct the (reduced) context containing that set of substituands.

That is, if A represents a λ-term M with substituands P_{i_j}, $i_j \in S$ such that:

$$\begin{aligned} [\![\mathit{LeftInterp}(A)]\!]_\lambda &\equiv M \\ &\equiv Q[z_{i_1} := P_{i_1}, \ldots, z_{i_n} := P_{i_n}],\ i_j \in S \end{aligned}$$

and $A \twoheadrightarrow A'$, then

$$[\![\mathit{Interp}_S(A)]\!]_\lambda \equiv Q'[z_{i_1} := P'_{i_1}, \ldots, z_{i_n} := P'_{i_n}],\ i_j \in S$$

is a substitution instance of the original context Q. Furthermore, Λ^{inc} computes the projection of a reduction on *every* such context simultaneously.

9 Other Issues

rwhnf() and Λ^{inc} have a number of additional interesting properties that we are able to mention here only in passing:

- *rwhnf*() performs reductions that are strongly equivalent to O'Donnell's *non-copying* reductions [O'D77], which are an efficient (though not optimal) class of parallel λ-reductions. This leads us to conclude that *rwhnf*() performs well (in terms of total number of redexes contracted) even in a non-incremental setting.

- We can show that Λ^{inc} is *relatively optimal*, i.e., that a non-overlapping reduction of a common term of a set of similar terms yields the *shortest* (in terms of **(Beta)** contractions) subsequent reduction of any new term resulting from substituand replacement, relative to the reduction strategy used by *rwhnf*().

— We can extend the model of allowable subtree replacements to allow substituands in completely arbitrary contexts, not just those representable by substitution. For example, let $N[x] \equiv \lambda xy.x$. Then we might wish to replace x with y to yield $N[y] \equiv \lambda xy.y$, which is not representable by substitution since the substituand is a bound variable. We can perform reduction on contexts such as N without losing track of the proper bindings since **ΛCCL** allows "checkpoints" to be made in the substitution process, not just before β-reduction.

— There are some interesting connections between the idea of projection of a reduction on a term and the semantic notion of *stability*.

We also note that a number of enhancements can be made to the implementation of Λ^{inc}:

— We can extend the reduction system to allow strict predefined functions and atomic values (e.g., addition and integers).

— The Y combinator can be implemented using a cyclic graph.

— Numerous enhancements can be made to the implementation of $\Delta rwhnf()$ to avoid such anomalous overhead as repeated examination of terms already in *whnf*, and chains of Δ nodes with the same index.

The issues mentioned here are all covered in greater detail in [Fie90a].

10 Applications and Conclusions

Incremental λ-reduction can be used in any situation where an algorithm expressed functionally is applied to (or transformed to) a set of similar terms. Our model of incrementality requires that the possible points of λ-term replacement be designated in advance. While at first glance this might seem a severe restriction, we note that one can, if desired, specify *every* subterm of a λ-term as a possible point of replacement, (given the remark above about extensions of the replacement model to arbitrary contexts). However, the overhead required to maintain the attendant Δ node "checkpoints" is increased accordingly.

We envision incremental reduction as most useful in applications similar to that of Section 2. In such cases, the λ-terms themselves are not the primary objects of interest, but instead are used to specify the "semantics" (in an operational way) of other languages, systems, or structured data, and are thus manipulated indirectly.

We also feel that the generality of incremental λ-reduction makes it a good starting point for devising techniques for incremental computation in more restricted domains.

To summarize, we have defined a notion of *incremental reduction* in the λ-calculus. The concept of *overlapping* reductions on similar terms is presented, and a practical incremental evaluator, Λ^{inc} is described. The reductions performed by Λ^{inc} on a sequence of terms are guaranteed to be non-overlapping, which we take as our definition of a (relatively) optimal incremental evaluator.

A The Lambda Calculus and Term Rewriting Systems

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

A.1 Notation

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

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

β-contraction is denoted by $\longrightarrow_\beta$. The reflexive, transitive closure of $\longrightarrow_\beta$, *β-reduction*, is denoted by $\twoheadrightarrow_\beta$. β-equality will be denoted by '=', syntactic identity of λ-terms (modulo renaming of bound variables) by '$\equiv$'.

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.

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

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

σ designates the entire reduction sequence. The *concatenation* of two reductions σ and ρ is denoted by $\sigma + \rho$, and is defined only if the final term of σ is the initial term of ρ. $\emptyset$ denotes the null reduction (the reduction of no terms).

One can define a notion of *parallel β-contraction*, notated $M \longrightarrow_{||\beta} N$, in which *sets* of redexes are contracted in a single step; see [Lév78,Lév80] for the formal details. Parallel *reductions* are represented thus:

$$\sigma\colon M_0 \xrightarrow{C_1}{}_{||\beta} M_1 \xrightarrow{C_2}{}_{||\beta} \cdots \xrightarrow{C_n}{}_{||\beta} M_n$$

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

The de Bruijn λ-calculus [dB72] 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.

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

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

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

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. Substitution and β-reduction can be suitably redefined on λ^{DB} to mimic the analogous operations in the λ-calculus. See [Cur86a] for a concise summary of the operational behavior of the de Bruijn λ-calculus.

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

B ΛCCL

B.1 Term Structure

Definition B.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.

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

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

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

Definition B.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 $\mathbf{\Lambda CCL}$ 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 $\mathbf{\Lambda CCL}$ 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 $\mathbf{\Lambda CCL}$ 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:

B.2 Axioms

Definition B.3 *The axioms of* **ΛCCL** *are as follows:*

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

B.3 ΛCCL as 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 $B \equiv C[Y]$.

B.4 Translation

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

Definition B.4 *For any term* $M \in \lambda^{\mathrm{DB}}$*, we can define a corresponding term* $[\![M]\!]_{\Lambda\mathrm{CCL}} \in$ **ΛCCL** *inductively as follows:*

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

B.5 Equivalence

There is an equivalence between β-reduction and reduction of terms of sort $\mathcal{L}$ in **ΛCCL**:

Theorem B.1 *Given $M \in \mathrm{Ter}(\lambda^{\mathrm{DB}})$,*

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

This result shows that *any* reduction of a **ΛCCL** term $A \in \mathit{LNF}$ simulates a reduction in the λ-calculus. The result above can be strengthened as follows:

Theorem B.2 *For all $A : \mathcal{L}$, $B : \mathcal{L}$, such that*

$$\rho\colon A \twoheadrightarrow_{\Lambda\mathrm{CCL}} B$$

there exists

$$\beta(\rho)\colon [\![A]\!]_\lambda \twoheadrightarrow_{||\beta} [\![B]\!]_\lambda$$

such that the number of (parallel) β-contractions in $\beta(\rho)$ equals the number of **(Beta)** *contractions in ρ.*

In other words, an arbitrary **ΛCCL** reduction can be simulated by a parallel β-reduction of the same length (using the number of **(Beta)** contractions as a length measure in the former case).

B.6 Normal Forms

Definition B.5 *The set of* lambda normal forms *(LNF) is a subset of the terms of* **ΛCCL**, *defined inductively as follows:*

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

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

The following fact makes possible a straightforward translation from terms in *LNF* to terms of the de Bruijn λ-calculus:

Theorem B.3 *All lambda-like expressions (terms of sort $\mathcal{L}$) of* **ΛCCL** *are reducible to a lambda normal form, using the rules of* **ΛCCL**; *i.e., for all $A\colon \mathcal{L}$, there exists $B \in \mathit{LNF}$ such that $A \twoheadrightarrow_{\Lambda\mathrm{CCL}} B$.*

The lambda normal form of A is given by $\mathit{lnf}(A)$. The translation $[\![A]\!]_\lambda$ is then defined as the inverse of $[\![\cdot]\!]_{\Lambda\mathrm{CCL}}$ on $\mathit{lnf}(A)$.

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.

[ACR+87] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. Incremental evaluation of attributed graphs. Technical Report RC 13205, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, October 1987.

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

[Ben82] S. A. Bengelloun. *Aspects of Incremental Computing.* PhD thesis, Yale University, 1982.

[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.

[BPR85] A. M. Berman, M. C. Paull, and B. G. Ryder. Proving relative lower bounds for incremental algorithms. Technical Report DCS-TR-154, Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903, 1985.

[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.

[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.

[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.

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. *Proc. Eighth ACM Symp. on Principles ofC Programming Languages*, pages 105–116, 1981.

[Ear76] J. Earley. High level iterators and a method for automatically designing data structure representation. *Journal of Computer Languages*, 1:321–342, 1976.

[Fie90a] John Field. *Incremental Reduction and its Applications.* PhD thesis, Cornell University, 1990. (Forthcoming).

[Fie90b] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, January 1990.

[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.

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

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2, 1989.

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

[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.

[O'D77] Michael J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag, 1977. Lecture Notes in Computer Science 58.

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

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[PT89] W. Pugh and T. Teitelbaum. Incremental computation by function caching. In *Proc. Sixteenth ACM Symp. on Principles of Programming Languages*, 1989.

[Rep82] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. *Proc. Ninth ACM Symp. on Principles of Programming Languages*, pages 169–176, 1982.

[Sun90] R.S. Sundaresh. Technical report. Yale University, 1990.

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

[YS87] D. Yellin and R. Strom. "INC": A language for incremental computations. Technical Report RC 13327, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, December 1987.