

CS-1994-03

**A Notation for Lambda Terms I: A
Generalization of Environments**

Gopalan Nadathur

Debra Sue Wilson

Department of Computer Science

Duke University

Durham, North Carolina 27708-0129

January 1994

A Notation for Lambda Terms I: A Generalization of Environments*

Gopalan Nadathur and Debra Sue Wilson

Department of Computer Science
Duke University, Durham, NC 27706
(gopalan@cs.duke.edu, dsw@cs.duke.edu)

Abstract

A notation for lambda terms is described that is useful in contexts where the intensions of these terms need to be manipulated. This notation uses the scheme of de Bruijn for eliminating variable names, thus obviating α -conversion in comparing terms. This notation also provides for a class of terms that can encode other terms together with substitutions to be performed on them. The notion of an environment is used to realize this ‘delaying’ of substitutions. The precise mechanism employed here is, however, more complex than the usual environment mechanism because it has to support the ability to examine subterms embedded under abstractions. The representation presented permits a β -contraction to be realized via an atomic step that generates a substitution and associated steps that percolate this substitution over the structure of a term. The operations on terms that are described also include ones for combining substitutions so that they might be performed simultaneously. Our notation eventually provides a basis for efficient realizations of β -reduction and also serves as a means for interleaving steps inherent in this operation with steps in other operations such as higher-order unification. The various manipulations on terms in our notation are described through a system of rewrite rules. The correspondence of this rewrite system to the usual notion of β -reduction is exhibited and is used in establishing confluence and other properties pertaining to it. The notation presented here is similar in spirit to other recent proposals deriving from the Categorical Combinators of Curien, and the relationship to these is discussed. Refinements to our notation and rewrite system and their use in describing manipulations on lambda terms are considered in detail in a companion paper.

1 Introduction

This paper concerns a notation for the terms in a lambda calculus that can serve as a basis for efficient implementations of operations on such terms. Traditionally, lambda terms have been used as a vehicle for performing computations, and the representation of these terms and the design of efficient evaluators for the lambda calculus in this context have received considerable attention. Our interest, however, is in a situation where lambda terms are used as a *representational* device. This interest is motivated primarily by implementation questions pertaining to λ Prolog,

* This paper has been submitted for publication. Comments on its contents are welcome and may be sent to the authors at the indicated addresses.

a logic programming language that employs the terms of a typed lambda calculus as its data structures [NM88]. We believe, however, that this issue is of wider concern, given the number of computer systems and programming languages in existence today that use some variety of the lambda calculus in representing and manipulating formal objects such as formulas, programs and proofs [Bru80, CAB⁺86, CH88, GMW79, HHP93, MN87, Pau88, Pfe89].

Lambda terms have been found to be useful as data structures because of their ability to represent naturally the notion of binding that is part of the syntax of several kinds of objects [Chu40, HL78, MN87, Pau87, PE88]. Consider, for instance, the task of representing the quantified formula $\forall x((p\ x) \vee (q\ x))$ in which p and q are predicate names. Observing that a quantifier plays the dual role of determining a scope and of making a predication, this formula can be rendered fairly transparently into the lambda term $(all\ (\lambda x((p\ x)\ or\ (q\ x))))$; in this term, all is a constant that represents universal quantification and or is an (infix) constant representing disjunction. Using such a representation makes the implementation of several logical operations on formulas relatively straightforward. For example, consider the operation of instantiation. Under the chosen representation, instantiating a ‘formula’ of the form $(all\ P)$ by t is given simply by the term $(P\ t)$. The actual task of substitution is carried out with all the necessary renamings by the β -reduction operation on lambda terms. As another example, consider the task of determining if a given formula has a certain structure; such an operation would be relevant, for instance, to the construction of a theorem prover. The notion of unifying lambda terms provides a powerful tool for performing such ‘template matching’. Thus, consider the term $(all\ (\lambda x((P\ x)\ or\ (Q\ x))))$ in which P and Q are variables. This term matches with any formula whose top-level structure is that of a universal quantification over a disjunction and thus ‘recognizes’ such formulas. In contrast, the term $(all\ (\lambda x((P\ x)\ or\ Q)))$ requires also that the second disjunct not contain the quantified variable and thus serves as a sharper discriminator.¹

Our interest in this paper is in a suitable representation for lambda terms, assuming that they are to be used in the manner outlined above. The intended application obviously places constraints on the kinds of representations that might be considered. For example, the applications of interest generally require the comparison of the structures of lambda terms. The chosen representation must therefore make this structure readily available. At a more detailed level, the comparison of lambda terms must ignore the particular names used for bound variables. To cater to this need, the representation that is used must permit equality up to α -convertibility to be determined easily. Finally, an operation of obvious importance is β -reduction, and any reasonable representation must enable this to be performed efficiently. For reasons that we discuss in Section 4, the representation that is used must permit two constraints to be satisfied by an implementation of this operation: first, it should be possible to perform the substitutions generated by β -contractions in a *lazy* manner and, second, it should be possible to perform β -contractions *under* abstractions as well as to percolate substitutions generated by it into such contexts.

We describe a notation for lambda terms in this paper that provides a basis for meeting these

¹The notion of unification (used in an informal sense here) is intelligible only in the context of certain typed versions of the lambda calculus. We do not discuss the issue of typing explicitly here since the main concerns of this paper are orthogonal to it.

various requirements. The starting point for our notation is a scheme suggested by de Bruijn [Bru72] for eliminating variable names from terms. To provide a means for delaying substitutions, we utilize the notion of an environment. However, a direct use of this device as developed in the context of implementations of functional programming languages is not possible; the complicating factor is the need for performing substitutions and β -contractions under abstractions. The notation we describe embellishes the notion of an environment in a manner designed to overcome this difficulty. At a level of detail, our proposal shares features with the data structures used in [AP81] in implementing a normalization procedure. However, in a manner akin to other recent proposals deriving from the Categorical Combinators of Curien [ACCL90, Cur86a, Fie90], it has the characteristic of reflecting the idea of an environment into the notation itself. There are two advantages to adopting this course. First, the resulting notation is fine-grained enough to support a wide variety of reduction procedures on lambda terms, and the analysis undertaken here makes it easy to verify the correctness of these procedures. Second, using such a notation makes it possible to intermingle what are traditionally conceived of as steps within β -contraction with other operations such as those needed in higher-order unification [Hue75]. There is, in fact, a concrete realization of the second idea: the notation developed here is actually being used in this fashion in an implementation of λ Prolog [NJW93].

The remainder of this paper is organized as follows. In the next section, we summarize the logical notions, particularly those pertaining to rewrite systems, that are used throughout this paper. Section 3 reviews the de Bruijn notation for lambda terms. We describe our notation for lambda terms in Section 4 and also present the rewrite rules that are intended to mimic β -reduction in its context. We then study the properties of our notation. In Section 5 we describe a comparison relation on our terms and show that it constitutes a well founded partial ordering relation. This relation is used later in establishing termination properties of subsets of our rules as well as in constructing inductive arguments. In the following section, we analyze a particular subset of our rewrite rules whose purpose is, roughly, that of reducing terms in our notation that encapsulate substitutions into ones in de Bruijn's notation. In particular, we show that any sequence of rewritings using these rules must terminate and must eventually produce the same de Bruijn term from a given term in our notation. In Section 7, we examine the correspondence between the usual notion of β -reduction and our system of rewrite rules. We show here that every β -reduction sequence on de Bruijn terms can be mimicked within our notation. We also demonstrate that any rewrite sequence on our terms can be projected onto an β -reduction sequence on the underlying de Bruijn terms. The advantage of our notation can then be appreciated as follows: it defines a notion of β -contraction that is a truly atomic operation and it provides a fine-grained control over the substitution process. In Section 8, we establish the confluence of our rewrite system. We utilize the 'projection' onto de Bruijn terms made possible by the results of Section 7 in showing this property. This idea is similar in spirit to that referred to as the *interpretation method* in [Har89] and used in [Har89] and [Yok89] in proving confluence properties of a combinator calculus. We conclude this paper in Section 9 by discussing the relationship of our work to that of others, especially that in [ACCL90] and [Fie90].

2 Logical preliminaries

We are concerned in this paper with systems for rewriting expressions. Each such rewrite system is specified by a set of rule schemata. A rule schema has the form $l \rightarrow r$ where l and r are expression schemata referred to as the lefthand side and the righthand side of the rule schema, respectively. For example, the system we describe in Section 4 contains the schemata:

$$\begin{aligned} ((\lambda t_1) t_2) &\rightarrow \llbracket t_1, 1, 0, (t_2, 0) :: nil \rrbracket, \quad \text{and} \\ \llbracket (t_1 t_2), ol, nl, e \rrbracket &\rightarrow (\llbracket t_1, ol, nl, e \rrbracket \llbracket t_2, ol, nl, e \rrbracket). \end{aligned}$$

In these schemata, t_1 , t_2 , ol , nl and e represent metalanguage variables ranging over appropriately defined categories of expressions. Particular rules may be obtained from these schemata by suitably instantiating these variables. All our rule schemata satisfy the property that any syntactic variable appearing in the righthand side already appears in the lefthand side.

Given a notion of subexpressions within the relevant expression language, a rule schema defines a relation between expressions as follows: t_1 is related to t_2 by the rule schema if t_2 is the result of replacing some subexpression s_1 of t_1 by s_2 where $s_1 \rightarrow s_2$ is an instance of the schema. We refer to occurrences in expressions of instances of the lefthand side of a rule schema as *redex* occurrences of the schema. The qualification by the rule schema may be omitted if it is clear from the context. Alternatively, the correspondence to the rule schema may be signified by the use of a special name; *e.g.*, see the name ‘ β -redex’ introduced in Section 3.

We refer to the relation corresponding to a rule schema as the one that is *generated* by it. The relation generated by a collection of rule schemata is the union of the relations generated by each schema in the collection. Let \triangleright denote such a relation. We will usually write \triangleright in infix form, *i.e.*, the fact that t is related to s by virtue of \triangleright will be expressed by writing $t \triangleright s$. The reflexive and transitive closure of \triangleright will be denoted by \triangleright^* , a relation that will, once again, be written in infix form. Intuitively, $t \triangleright^* s$ signifies that t can be rewritten to s by a (possibly empty) sequence of applications of the relevant rule schemata. In accordance with this viewpoint, we refer to the relation \triangleright as a *rewrite* or *reduction* relation and we say that $t \triangleright$ -reduces to s if $t \triangleright^* s$.

A notion of concern with regard to a rewrite relation \triangleright is that of a \triangleright -*normal form*. An expression t is in this form if there is no expression s such that $t \triangleright s$. That is, t contains no redex occurrences of any of the rule schemata that generate \triangleright . A \triangleright -normal form *of* an expression r is an expression t such that $r \triangleright$ -reduces to t and t is in \triangleright -normal form. The existence and uniqueness of normal forms for expressions are issues that are of interest for a variety of reasons. For example, rewrite rules are often used as a means for computing. Their use in this capacity is meaningful only if the result of performing the computation — the normal form, if it exists — is independent of the method of carrying out the computation. This will be the case if normal forms are unique. In a sense more pertinent to this paper, a collection of rewrite rule schemata is usually intended as a set of equality axioms in a given logical system. Using them to rewrite expressions is useful in this context only if this somehow helps in determining equality. This is indeed the case if a unique normal form exists for every expression: the equality of two expressions can then be determined by reducing them to their normal forms and comparing these.

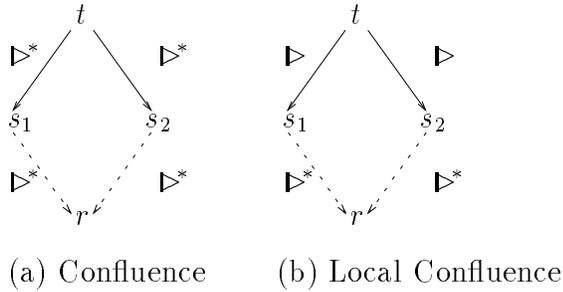


Figure 1: Diagrammatic Depiction of Confluence and Local Confluence

A rewrite relation \triangleright is *noetherian* if and only if there is no infinite sequence of the form $t_1 \triangleright t_2 \triangleright \dots \triangleright t_n \triangleright \dots$, *i.e.*, if and only if every sequence of rewritings relative to \triangleright terminates. If \triangleright is noetherian, a \triangleright -normal form must exist for every expression. In showing that such a form is unique, the notion of *confluence* is useful. The relation \triangleright is said to be confluent if, given any expressions t , s_1 and s_2 such that $t \triangleright^* s_1$ and $t \triangleright^* s_2$, there must be some expression r such that $s_1 \triangleright^* r$ and $s_2 \triangleright^* r$. We express the fact that this property holds by the picture in Figure 1(a); a dashed arrow in a picture of this kind signifies the existence of a reduction given by the label on the arrow, depending on the reductions depicted by the solid arrows. Confluence is of interest because of the following proposition whose proof is straightforward.

Proposition 2.1 *If \triangleright is a confluent reduction relation, then if a \triangleright -normal form exists for any expression, it must be unique.*

There is a notion weaker than confluence called *local confluence* that is also of interest. A rewrite relation \triangleright is said to be locally confluent if, whenever $t \triangleright s_1$ and $t \triangleright s_2$ for expressions t , s_1 and s_2 , there must be some expression r such that $s_1 \triangleright^* r$ and $s_2 \triangleright^* r$. The fact that such a property holds is expressed pictorially as in Figure 1(b). Local confluence is related to confluence by the following proposition, a proof for which may be found in [Hue80].

Proposition 2.2 *A noetherian reduction relation is confluent if and only if it is locally confluent.*

To show that a reduction relation \triangleright is locally confluent, we need to do the following. For each expression t , we need to consider all the expressions that result from rewriting some subexpression of t using the rule schemata. Then, we need to show that each pair of these expressions can be reduced by a sequence of rewritings to a common expression. This task may be simplified by using an observation in [KB70] that is generalized in [Hue80]. To describe this observation, we need the following definition.

Definition 2.3 An expression t constitutes a *nontrivial overlap* of rule schemata R_1 and R_2 at a subexpression s of t if (a) t is a redex occurrence of R_1 , (b) s is a redex occurrence of R_2 and also does not instantiate a schema variable when t is matched with R_1 , and (c) either s is distinct from

t or R_1 is distinct from R_2 . Let r_1 be the expression that results from rewriting t using R_1 and let r_2 result from t by rewriting s using R_2 . Then the pair $\langle r_1, r_2 \rangle$ is referred to as the *conflict pair* relative to the overlap in question. The conflict pairs of a collection of rule schemata \mathcal{R} is the set of the conflict pairs obtained by considering all possible nontrivial overlaps between the elements of \mathcal{R} .

The conflict pairs as defined here constitute all the ground instances of the critical pairs of a rewrite system in the sense of [Hue80]. We use the notion of critical pairs only at a metalanguage level to avoid a consideration of expressions containing variables.

The observation that is critical to showing local confluence is now the following:

Theorem 2.4 *Let \triangleright be a reduction relation generated by the collection \mathcal{R} of rule schemata. Then \triangleright is locally confluent if and only if for every conflict pair $\langle r_1, r_2 \rangle$ of \mathcal{R} there is some expression s such that $r_1 \triangleright^* s$ and $r_2 \triangleright^* s$.*

Proof. (Adapted from [Hue80].) Since $\langle r_1, r_2 \rangle$ is a conflict pair, there is some t such that $t \triangleright r_1$ and $t \triangleright r_2$. Thus, if \triangleright is locally confluent, there must be some s such that $r_1 \triangleright^* s$ and $r_2 \triangleright^* s$. This yields the ‘only if’ part.

We now consider the ‘if’ part. Let t be any expression and let t_1 and t_2 be the result of rewriting, respectively, the subexpressions s_1 and s_2 in t using the members R_1 and R_2 of \mathcal{R} . To show that \mathcal{R} is locally confluent, we need to show that there is some expression r such that $t_1 \triangleright^* r$ and $t_2 \triangleright^* r$. We consider the various possibilities for s_1 and s_2 and show that this must be the case. If s_1 and s_2 appear in disjoint parts of t , this is obvious: there is a ‘residue’ of s_2 in t_1 and similarly of s_1 in t_2 and a common expression is obtained by rewriting the first of these (in t_1) using R_2 and the second (in t_2) using R_1 . So suppose that one of s_1 and s_2 is a subexpression of the other. Without loss of generality, let s_2 be a subexpression of s_1 . Now, if s_1 is identical to s_2 and $R_1 = R_2$, then $t_1 = t_2$ and the desired conclusion is immediately reached. If s_2 is a subexpression of a part of s_1 that is matched with a schema variable in R_1 , a little additional argument suffices. On the one hand, the rewriting step that produces t_1 will create a finite number of copies of s_2 in t_1 and on the other hand rewriting s_2 produces in t_2 a subexpression s'_1 that is still a redex occurrence of R_1 . It is easily seen that using R_2 repeatedly to rewrite the copies of s_2 in t_1 and R_1 to rewrite s'_1 in t_2 produces a common expression. The only remaining situation is the one where s_2 is a subexpression of s_1 that matches with a part of R_1 distinct from a schema variable and where either s_1 is distinct from s_2 or R_1 is distinct from R_2 . However, in this case s_1 constitutes a nontrivial overlap of R_1 and R_2 at s_2 . Let r_1 result from rewriting s_1 using R_1 and let r_2 result from s_1 by rewriting the subexpression s_2 using R_2 . Then $\langle r_1, r_2 \rangle$ constitutes a conflict pair of \mathcal{R} and, by assumption, there is an expression s such that $r_1 \triangleright^* s$ and $r_2 \triangleright^* s$. Let r be the expression obtained from t by replacing the subexpression s_1 by s . It must then be the case that $t_1 \triangleright^* r$ and $t_2 \triangleright^* r$.

□

3 The de Bruijn notation

Conventional presentations of the lambda calculus utilize a scheme that requires names for bound (and free) variables (*e.g.* see [HS86]). This choice is well-motivated from the perspective of human readability but it is not well-suited to machine implementations for at least two reasons. First, care needs to be exercised during β -reduction to prevent inadvertent capture of free variables. Second, the determination of identity of two terms is complicated by the need to consider renamings for bound variables. These problems are eliminated by the ‘nameless’ notation proposed by de Bruijn [Bru72] which provides what might be thought of as the ‘right’ abstract syntax for lambda terms. The de Bruijn notation is central to the discussions in this paper and we therefore outline it briefly below. We begin by defining a term in this representation:

Definition 3.1 The collection of *de Bruijn terms*, denoted by the syntactic category $\langle DTerm \rangle$, is given by the rule

$$\langle DTerm \rangle ::= \langle Cons \rangle \mid \# \langle Index \rangle \mid (\langle DTerm \rangle \langle DTerm \rangle) \mid (\lambda \langle DTerm \rangle)$$

where $\langle Cons \rangle$ is a category corresponding to a predetermined set of constant symbols and $\langle Index \rangle$ is the category of positive numbers. A de Bruijn term of the form (i) $\#i$ is referred to as an *index* or a *variable reference*, (ii) (λt) is called an *abstraction* and (iii) $(t_1 t_2)$ is referred to as an *application*. The subterm or subexpression relation on de Bruijn terms is given recursively as follows: Each term is a subterm of itself. If t is of the form $(\lambda t')$, then each subterm of t' is also a subterm of t . If t is of the form $(t_1 t_2)$, then each subterm of t_1 and of t_2 is also a subterm of t .

A bound variable occurrence within the conventional scheme for writing lambda terms is represented in the de Bruijn notation by an index that counts the number of abstractions between the occurrence and the abstraction binding it. Thus, the term $(\lambda x ((\lambda y (y x)) x))$ in conventional presentations is written in the de Bruijn notation as $(\lambda ((\lambda (\#1 \#2)) \#1))$. An alternative, more complete, exposition of the correspondence is the following. We think of the *level* of a subterm in a term as the number of abstractions in the term within which the subterm is embedded. Further, we assume a fixed listing of the free variables with respect to which we can talk of the n -th free variable. Then, a variable reference $\#i$ occurring at level j in a term corresponds to a bound variable if $j \geq i$. Further, in this case, it represents a variable that is bound by the abstraction at level $(j - i)$ within which the variable reference occurs. In the case that $i > j$, the index $\#i$ represents a free variable, and, in fact, the $(i - j)$ -th free variable. We observe that lambda terms that are α -convertible in the conventional notation correspond to terms that are identical in this notation. For example, the terms $(\lambda x ((\lambda y (y x)) x))$ and $(\lambda z ((\lambda w (w z)) z))$ both correspond to the same de Bruijn term, $(\lambda ((\lambda (\#1 \#2)) \#1))$. Thus, a virtue of the de Bruijn notation from our perspective is that it obviates α -conversion.

An important operation on lambda terms is that of substitution. In the context of the de Bruijn notation, a generalized notion of substitution — that of substituting terms for *all* the free variables — is given by the following definition.

Definition 3.2 Let t be a de Bruijn term and let s_1, s_2, s_3, \dots represent an infinite sequence of de

Bruijn terms. Then the result of simultaneously substituting s_i for the i -th free variable in t for $i \geq 1$ is denoted by $S(t; s_1, s_2, s_3, \dots)$ and is defined recursively as follows:

- (1) $S(c; s_1, s_2, s_3, \dots) = c$, for any constant c ,
- (2) $S(\#i; s_1, s_2, s_3, \dots) = s_i$ for any variable reference $\#i$,
- (3) $S((t_1 t_2); s_1, s_2, s_3, \dots) = (S(t_1; s_1, s_2, s_3, \dots) S(t_2; s_1, s_2, s_3, \dots))$, and
- (4) $S((\lambda t); s_1, s_2, s_3, \dots) = (\lambda S(t; \#1, s'_1, s'_2, s'_3, \dots))$ where, for $i \geq 1$, $s'_i = S(s_i; \#2, \#3, \#4, \dots)$.

We shall use the expression $S(t; s_1, s_2, s_3, \dots)$ as a meta-notation for the term it denotes.

Towards understanding the above definition, we note that within a term of the form (λt) , the first free variable is actually denoted by the index $\#2$, the second by $\#3$ and so on. This requires, in (4) above, that the indices for free variables within the terms s_1, s_2, s_3, \dots being substituted into (λt) be “incremented” by 1 prior to substitution into t . Further, the index $\#1$ must remain unchanged within t and it is the indices $\#2, \#3, \dots$ that must be substituted for.

We will need to consider the effect of cascading substitutions of the above kind. An observation made in [Bru72] is useful in this context. The term denoted by $S(S(t; s_1, s_2, s_3, \dots); s'_1, s'_2, s'_3, \dots)$ is produced by first replacing *every* index in t with some term s_i , and then substituting the terms s'_1, s'_2, s'_3, \dots into the result. Thus the s'_i terms will only be substituted into occurrences of the s_j terms, and the effect of this substitution can be precomputed. This is formalized in the following proposition taken from [Bru72].

Proposition 3.3 *Given de Bruijn terms $t, s_1, t_1, s_2, t_2, s_3, t_3 \dots$*

$$S(S(t; s_1, s_2, s_3, \dots); t_1, t_2, t_3, \dots) = S(t; u_1, u_2, u_3, \dots)$$

where, for $i \geq 1$, $u_i = S(s_i; t_1, t_2, t_3, \dots)$.

The above substitution operation is useful in defining the notion of \triangleright_β -reduction, also referred to simply as β -reduction.

Definition 3.4 The β -contraction rule schema is the following

$$((\lambda t_1) t_2) \rightarrow S(t_1; t_2, \#1, \#2, \dots)$$

where t_1 and t_2 are schema variables for de Bruijn terms. The relation (on de Bruijn terms) generated by this rule schema is denoted by \triangleright_β and is called β -contraction. An instance of the lefthand side of the rule schema is called a β -redex.

When a β -contraction is performed, the β -redex is replaced by the term which results from substituting t_2 for the first free variable in t_1 and adjusting the remaining indices. In the next section a notation will be introduced which decouples the generation and performance of the substitution by, in essence, moving the meta-notation $S(t_1; t_2, \#1, \#2, \dots)$ into the term representation. The following theorem states a property of commutativity between β -reduction and the substitution operation that will be useful in analyzing this notation.

Theorem 3.5 *Let t_0, t_1, t_2, \dots be de Bruijn terms.*

- (i) *If $t_0 \triangleright_{\beta}^* t'_0$, then $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta}^* S(t'_0; t_1, t_2, t_3, \dots)$.*
- (ii) *If, for $i \geq 1$, $t_i \triangleright_{\beta}^* t'_i$, then $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta}^* S(t_0; t'_1, t'_2, t'_3, \dots)$*

Proof. (i) It suffices to show that if $t_0 \triangleright_{\beta} t'_0$ then $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta} S(t'_0; t_1, t_2, t_3, \dots)$. We do this by an induction on the structure of t_0 . Note first that t_0 must be either an abstraction or an application. Suppose t_0 is an abstraction. In particular, let $t_0 = (\lambda s)$. Then the redex that is rewritten must be a subterm of s . The desired conclusion now follows from Definition 3.2 and the inductive hypothesis. If t_0 is an application, there are two possibilities. In the first case, t_0 is not the redex rewritten. In this case we again use Definition 3.2 and the inductive hypothesis to reach the desired conclusion. In the other case, let $t_0 = ((\lambda s_1) s_2)$. Then $t'_0 = S(s_1; s_2, \#1, \#2, \dots)$. Now,

$$S(t_0; t_1, t_2, t_3, \dots) = ((\lambda S(s_1; \#1, t'_1, t'_2, \dots)) S(s_2; t_1, t_2, t_3, \dots))$$

where, for $i \geq 1$, $t'_i = S(t_i; \#2, \#3, \#4, \dots)$. But then

$$S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta} S(S(s_1; \#1, t'_1, t'_2, \dots); S(s_2; t_1, t_2, t_3, \dots), \#1, \#2, \dots)$$

Using Proposition 3.3,

$$\begin{aligned} S(S(s_1; \#1, t'_1, t'_2, \dots); S(s_2; t_1, t_2, t_3, \dots), \#1, \#2, \dots) = \\ S(s_1; S(s_2; t_1, t_2, t_3, \dots), t''_1, t''_2, \dots). \end{aligned}$$

where $t''_i = S(t'_i; S(s_2; t_1, t_2, t_3, \dots), \#1, \#2, \dots)$. Noting the definition of t'_i and using Proposition 3.3 again, it can be seen that $t''_i = t_i$. Thus, $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta} S(s_1; S(s_2; t_1, t_2, t_3, \dots), t_1, t_2, \dots)$. On the other hand,

$$\begin{aligned} S(t'_0; t_1, t_2, t_3, \dots) &= S(S(s_1; s_2, \#1, \#2, \dots); t_1, t_2, t_3, \dots) \\ &= S(s_1; S(s_2; t_1, t_2, t_3, \dots), t_1, t_2, \dots) \end{aligned} \quad (\text{Proposition 3.3}).$$

Thus, even in this case, $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta} S(t'_0; t_1, t_2, t_3, \dots)$.

(ii) The proof is again by induction on the structure of t_0 . The constant and index cases are immediate and the application case is handled by a straightforward recourse to Definition 3.2 and the inductive hypothesis. The only remaining case is that when t_0 is of the form (λs) . In this case

$$S(t_0; t_1, t_2, t_3, \dots) = (\lambda S(s; \#1, u_1, u_2, \dots))$$

where $u_i = S(t_i; \#2, \#3, \#4, \dots)$. By (i), for $i \geq 1$, $u_i \triangleright_{\beta}^* S(t'_i; \#2, \#3, \#4, \dots)$. Using the inductive hypothesis and Definition 3.2, it now follows easily that $S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta}^* S(t_0; t'_1, t'_2, t'_3, \dots)$ □

The following corollary is proved by using Theorem 3.5 twice.²

²This corollary generalizes a theorem in [Bru72] that is used in proving the Church-Rosser Theorem for β -reduction.

Corollary 3.6 *Let t_0, t_1, t_2, \dots be de Bruijn terms and, for $i \geq 0$, let $t_i \triangleright_{\beta}^* t'_i$. Then*

$$S(t_0; t_1, t_2, t_3, \dots) \triangleright_{\beta}^* S(t'_0; t'_1, t'_2, t'_3, \dots).$$

Finally, we observe the celebrated Church-Rosser Theorem for β -reduction. A proof of it in the context of the de Bruijn notation appears in [Bru72].

Proposition 3.7 *The relation \triangleright_{β} is confluent.*

4 Incorporating environments into terms

The de Bruijn notation is useful in contexts where the intensions of lambda terms have to be examined because it makes it unnecessary to consider α -conversion. However, the operation of substitution necessitated by β -contraction is a fairly complex one even within this notation. From a practical perspective, it is useful to obtain some control over this substitution operation and, in particular, to be able to perform it lazily. For instance, consider the task of determining whether the two terms

$$((\lambda(\lambda(\lambda((\#3 \#2) s)))) (\lambda \#1)) \text{ and } ((\lambda(\lambda(\lambda((\#3 \#1) t)))) (\lambda \#1))$$

are equal modulo the rules of λ -conversion; s and t denote arbitrary terms here. It might be concluded that they are not, by observing that these terms reduce to $(\lambda(\lambda(\#2 s')))$ and $(\lambda(\lambda(\#1 t')))$, where s' and t' result from s and t by appropriate substitutions. Notice that it is enough to determine that the *heads* of these terms are distinct *without* explicitly performing the potentially costly operation of substitution on the arguments. Along a different direction, we observe that the structures of terms have to be traversed while attempting to reduce them to normal forms as well as in performing the substitutions generated by each β -contraction. The ability to delay substitutions enables these traversals to be combined, thereby leading to gains in efficiency. As an illustration, consider the term $((\lambda((\lambda t_1) t_2)) t_3)$ where t_1 , t_2 and t_3 represent arbitrary terms. Let t'_2 be the result of substituting t_3 for the ‘first’ free variable in t_2 and decrementing the indices of all the other free variables by one. Now, in reducing the given term to a normal form, it is necessary to substitute t'_2 and t_3 for the the first and second free variables in t_1 and to decrement the indices of all the other free variables by two. All these substitutions can be achieved in *one* traversal over the structure of t_1 provided we have a fine-grained control over the way each substitution is carried out. An observation of this kind is, in fact, exploited in the implementation of β -reduction in [AP81].

In contexts where the lambda calculus is used as a vehicle for computation, the notion of an environment that describes bindings for free variables suffices for delaying substitutions. In the situation where the de Bruijn notation is used, this device is adequate only because the structure of terms embedded within abstractions need not be explored. Thus, if a term is produced in the course of β -reduction that has an abstraction at the outermost level, then the term may be combined with its environment and returned as a *closure*; this idea is used, for instance, in [CCM87]. However, this assumption is *not* appropriate in contexts where lambda terms are used as a means for representation. As an example, consider again the task of determining whether the two terms

$$((\lambda (\lambda (\lambda ((\#3 \#2) s)))) (\lambda \#1)) \text{ and } ((\lambda (\lambda (\lambda ((\#3 \#1) t)))) (\lambda \#1))$$

are equal. In ascertaining that they are not, it is necessary to propagate a substitution generated by a β -contraction *under* an abstraction and also to contract β -redexes embedded *inside* abstractions. Examining the substitution operation now shows that the idea of an environment cannot be used in a straightforward fashion to yield a delaying mechanism in our context. For instance, if a term of the form $((\lambda t) s)$ is embedded within abstractions, it is to be expected that (λt) contains free variables. Hence, if the result of β -contracting this term is to be encoded by the term t and an ‘environment’, the environment must record not just the substitution of s for the first free variable but also the ‘decrementing’ of the indices corresponding to all the other free variables. Similar observations can be made about propagating substitutions under abstractions.

While the usual idea of an environment cannot be employed directly, a generalization of this notion suffices even in the context of interest. We describe a notation for lambda terms in this section that incorporates such a generalization into the de Bruijn representation for these terms. Our notation provides a means for capturing the generation of the substitution corresponding to a β -contraction in a truly atomic step. This operation is then combined with rules for ‘reading’ terms to realize the full effect of the complex substitution operation described in Section 3. We describe this notation below and then present a collection of rewrite rules on terms in the notation that serve to simulate β -reduction.

4.1 A modified syntax for terms

The main addition to the syntax of de Bruijn terms that yields our notation is that of a *suspension*. In an informal sense, a suspension is a quadruple consisting of a term together with two indices and an environment. The first index indicates an embedding level with respect to which variable references have been determined within the term, and the second index indicates a new embedding level. The environment determines bindings for a finite initial segment of the free variables. In its simplest form, an environment consists of a list of bindings. However, we permit environments of a more complex form in our terms; these, as we shall see later, are useful in ‘merging’ environments that eventually lead to the combination of substitutions. Formally, the syntax of the expressions of interest is given as follows:

Definition 4.1 The categories of *suspension terms*, *environments* and *environment terms*, denoted by $\langle STerm \rangle$, $\langle Env \rangle$ and $\langle ETerm \rangle$, are defined by the following syntax rules:

$$\begin{aligned} \langle STerm \rangle & ::= \langle Cons \rangle \mid \# \langle Index \rangle \mid ((\langle STerm \rangle \langle STerm \rangle) \mid \\ & \quad (\lambda \langle STerm \rangle) \mid \llbracket \langle STerm \rangle, \langle Nat \rangle, \langle Nat \rangle, \langle Env \rangle \rrbracket \\ \langle Env \rangle & ::= nil \mid \langle ETerm \rangle :: \langle Env \rangle \mid \{\langle Env \rangle, \langle Nat \rangle, \langle Nat \rangle, \langle Env \rangle\} \\ \langle ETerm \rangle & ::= @ \langle Nat \rangle \mid ((\langle STerm \rangle, \langle Nat \rangle) \mid \llbracket \langle ETerm \rangle, \langle Nat \rangle, \langle Nat \rangle, \langle Env \rangle \rrbracket). \end{aligned}$$

We assume that $\langle Cons \rangle$ and $\langle Index \rangle$ are as in Definition 3.1 and that $\langle Nat \rangle$ is the category of natural numbers. We refer to suspension terms, environments and environment terms collectively as *suspension expressions*.

We observe that the class of suspension terms includes all the de Bruijn terms. Conversely, there is only one new form for terms in the suspension notation: $\llbracket \langle STerm \rangle, \langle Nat \rangle, \langle Nat \rangle, \langle Env \rangle \rrbracket$. By an extension of terminology, we shall refer to suspension terms of the form $\#i$, (λt) and $(t_1 t_2)$ as indices or variable references, abstractions and applications, respectively. Terms of the new form are what are referred to as suspensions. An example of such a term is $\llbracket t_1, 1, 0, (t_2, 0) :: nil \rrbracket$, where t_1 and t_2 are de Bruijn terms. As we shall see presently, this term might arise from the ‘ β -contraction’ of $((\lambda t_1) t_2)$. It represents the fact that the term t_1 that was originally in the scope of 1 abstraction is now to be thought of as being in the scope of none and that t_2 , originally in the scope of 0 abstractions, is to be substituted for the first free variable in t_1 . Viewed differently, $\llbracket t_1, 1, 0, (t_2, 0) :: nil \rrbracket$ encodes the term that results from substituting t_2 for the first free variable in t_1 and decrementing the indices corresponding to the remaining free variables by 1. Such a term is called a suspension because it embodies a suspended substitution in the sense of Section 3. We will soon describe rules for rewriting terms that permit such suspended substitutions to be incrementally brought closer to completion.

The qualification ‘suspension’ applied to our terms and expressions is intended to distinguish them from similar notions in the context of the de Bruijn notation. We shall henceforth drop this qualification assuming that we are talking about terms and expressions in the new notation unless otherwise stated.

Definition 4.2 The *immediate subexpression(s)* of an expression x are given as follows:

- (1) If x is a term, then if (a) x is $(t_1 t_2)$, these are t_1 and t_2 , (b) if x is (λt) , this is t , and (c) if x is $\llbracket t, ol, nl, e \rrbracket$, these are t and e .
- (2) If x an environment, then (a) if x is $et :: e$, these are et and e , and (b) if x is $\{\{e_1, i, j, e_2\}\}$, these are e_1 and e_2 .
- (3) If x is an environment term, then (a) if x is (t, l) , then this is t , and (b) if x is $\langle\langle et, i, j, e \rangle\rangle$, then these are et and e .

The *subexpressions* of an expression are the expression itself and the subexpressions of its immediate subexpressions. We sometimes use the term *subterm* when the subexpression in question is a term. A *proper* subexpression of an expression x is any subexpression distinct from x .

We shall have occasion to talk about expressions whose environment subexpressions correspond transparently to a *list* of bindings. This class of expressions is identified by the following definition.

Definition 4.3 A *simple expression* is an expression that does not have subexpressions of the form $\langle\langle et, j, k, e \rangle\rangle$ or $\{\{e_1, j, k, e_2\}\}$. If the expression in question is a term, an environment or an environment term, it may be referred to as a simple term, a simple environment or a simple environment term, respectively. Note that a simple environment e is either *nil* or of the form $et_1 :: et_2 :: \dots :: et_n :: nil$. In the latter case, for $1 \leq i \leq n$, we write $e[i]$ to denote et_i ; observe that $e[i]$ must itself be of the form $@l$ or (t, l) . Further, for $1 \leq j \leq n$, we write $e\{j\}$ to denote the environment $et_j :: \dots :: et_n :: nil$.

Definition 4.4 The *length* of an environment e , denoted by $len(e)$, is given as follows: (a) if e is nil then $len(e) = 0$; (b) if e is $et :: e'$ then $len(e) = len(e') + 1$; and (c) if e is $\{\{e_1, i, j, e_2\}\}$ then $len(e) = len(e_1) + (len(e_2) \dot{-} i)$.

The symbol $\dot{-}$ used in the definition above denotes the subtraction operation on natural numbers. To understand this definition we note that, for a simple environment e , if $len(e) = 0$ then e must be nil and if $len(e) = n > 0$ then e must be of the form $et_1 :: et_2 :: et_3 :: \dots :: et_n :: nil$. Thus, this definition is motivated by the view of a simple environment e as a list of length $len(e)$.

Simple environment terms are of the form (t, i) or $@i$. Expressions of the former kind represent a substitution of t for an appropriately determined free variable. Expressions of the latter kind represent a ‘dummy’ binding, arising from the propagation of a substitution inside an abstraction. These expressions also encode the number of abstractions within whose scope the suspension term being substituted or the abstraction entered are to be thought to appear. This idea is made precise through the notion of an *index* that is defined below for all environment terms and is extended, for technical reasons, to environments as well.

Definition 4.5 The *index* of an environment term et , denoted by $ind(et)$, and, for each natural number l , the l -th index of an environment e , denoted by $ind_l(e)$, are defined simultaneously by structural induction on expressions as follows³:

- (i) If et is $@m$ then $ind(et) = m + 1$.
- (ii) If et is (t', m) then $ind(et) = m$.
- (iii) If et is $\langle\langle et', j, k, e \rangle\rangle$, let $m = (j \dot{-} ind(et'))$. Then

$$ind(et) = \begin{cases} ind_m(e) + (j \dot{-} k) & \text{if } len(e) > m \\ ind(et') & \text{otherwise.} \end{cases}$$

- (iv) If e is nil then $ind_l(e) = 0$.
- (v) If e is $et :: e'$ then $ind_0(e) = ind(et)$ and $ind_{l+1}(e) = ind_l(e')$.
- (vi) If e is $\{\{e_1, j, k, e_2\}\}$, let $m = (j \dot{-} ind_l(e_1))$ and $l_1 = len(e_1)$. Then

$$ind_l(e) = \begin{cases} ind_m(e_2) + (j \dot{-} k) & \text{if } l < l_1 \text{ and } len(e_2) > m \\ ind_l(e_1) & \text{if } l < l_1 \text{ and } len(e_2) \leq m \\ ind_{(l-l_1+j)}(e_2) & \text{if } l \geq l_1. \end{cases}$$

The index of an environment, denoted by $ind(e)$, is $ind_0(e)$.

We are eventually interested in only a subcollection of the suspension expressions that are referred to as the *well formed* ones. The wellformedness conditions that are presented below have the following properties: the class of suspension expressions that they deem to be well formed

³For environment terms and environments that are well formed in the sense of Definition 4.6, the $\dot{-}$ operation in the definitions of m that appear in items (iii) and (vi) in this definition may be replaced by simple subtraction.

includes the de Bruijn terms and is closed under the rewrite relation that we shall describe on these expressions.

Definition 4.6 An expression is *well formed* if the following conditions hold of every subexpression s of the expression:

- (i) If s is of the form $\llbracket t, ol, nl, e \rrbracket$ then $len(e) = ol$ and $ind(e) \leq nl$.
- (ii) If s is of the form $et :: e$ then $ind(e) \leq ind(et)$.
- (iii) If s is of the form $\langle\langle et, j, k, e \rangle\rangle$ then $len(e) = k$ and $ind(et) \leq j$.
- (iv) If s is of the form $\{\{e_1, j, k, e_2\}\}$ then $len(e_2) = k$ and $ind(e_1) \leq j$.

The following property of environments is useful in understanding the nature of the well-formedness constraints and in later analysis.

Lemma 4.7 *Let e be a well formed environment. Then $ind_i(e) \geq 0$. Further, for $i \geq len(e)$, $ind_i(e) = 0$. Finally, for any natural numbers i, j such that $i < j$, it is the case that $ind_i(e) \geq ind_j(e)$.*

Proof. By an induction on the structure of e . The proof of the first two properties is straightforward, and we only argue the truth of the last property.

In the case that e is *nil*, it is obvious that this property holds.

If e is of the form $et :: e'$, the hypothesis applied to e' easily yields the truth of the property for the case when $i \geq 1$. For the remaining case, we note that, by definition, $ind_0(e) = ind(et)$ and, because e is well formed, $ind(et) \geq ind_0(e') = ind_1(e)$. From this it is clear that the property holds even when $i = 0$.

Finally, suppose that e is of the form $\{\{e_1, l, m, e_2\}\}$. If $len(e_1) \leq i$, the truth of the property follows from applying the hypothesis to e_2 . In the case that $i < len(e_1)$, our analysis breaks up into two major subcases.

First assume that $len(e_2) > (l - ind_i(e_1))$. If $j < len(e_1)$ and $len(e_2) > (l - ind_j(e_1))$, the truth of the property follows from the hypothesis applied first to e_1 and then to e_2 . If $j < len(e_1)$ and $len(e_2) \leq (l - ind_j(e_1))$, since $m = len(e_2)$, it follows that $(l - m) \geq ind_j(e_1)$ and thus $ind_i(e) \geq ind_j(e)$. Finally, if $j \geq len(e_1)$, then $(l + j - len(e_1)) \geq l - ind_i(e_1)$ and then the hypothesis applied to e_2 verifies the truth of the property.

Now assume that $len(e_2) \leq (l - ind_i(e_1))$. If $j < len(e_1)$, applying the hypothesis to e_1 we see that $ind_i(e_1) \geq ind_j(e_1)$, and therefore $len(e_2) \leq (l - ind_j(e_1))$. Applying the hypothesis to e_1 then yields the property in this case. If $len(e_1) \leq j$, we observe first that $l \geq len(e_2)$ if $(l - ind_i(e_1)) \geq len(e_2)$. Thus, $(l + j - len(e_1)) \geq len(e_2)$ and so $ind_j(e) = 0$ and the property is again seen to be true.

□

$$(\beta_s) \quad ((\lambda t_1) t_2) \rightarrow \llbracket t_1, 1, 0, (t_2, 0) :: nil \rrbracket$$

Figure 2: The β_s -contraction rule schema

- (r1) $\llbracket c, ol, nl, e \rrbracket \rightarrow c$,
provided c is a constant.
- (r2) $\llbracket \#i, 0, nl, nil \rrbracket \rightarrow \#j$,
where $j = i + nl$.
- (r3) $\llbracket \#1, ol, nl, @l :: e \rrbracket \rightarrow \#j$,
where $j = nl - l$.
- (r4) $\llbracket \#1, ol, nl, (t, l) :: e \rrbracket \rightarrow \llbracket t, 0, nl', nil \rrbracket$,
where $nl' = nl - l$.
- (r5) $\llbracket \#i, ol, nl, et :: e \rrbracket \rightarrow \llbracket \#i', ol', nl, e \rrbracket$,
where $i' = i - 1$ and $ol' = ol - 1$, provided $i > 1$.
- (r6) $\llbracket (t_1 t_2), ol, nl, e \rrbracket \rightarrow (\llbracket t_1, ol, nl, e \rrbracket \llbracket t_2, ol, nl, e \rrbracket)$.
- (r7) $\llbracket (\lambda t), ol, nl, e \rrbracket \rightarrow (\lambda \llbracket t, ol', nl', @nl :: e \rrbracket)$,
where $ol' = ol + 1$ and $nl' = nl + 1$.

Figure 3: Rule schemata for reading suspensions

The content of Definition 4.6 may be appreciated at an intuitive level by considering a simple term of the form $\llbracket t, ol, nl, e \rrbracket$. If this term is well formed, it must be the case that $len(e) = ol$, *i.e.*, e must be of the form $et_1 :: et_2 :: \dots :: et_{ol} :: nil$. Further, for $1 \leq i \leq ol$, if et_i is of the form $@m$ then $m < nl$ and if et_i is of the form (t, m) then $m \leq nl$. Finally, from Lemma 4.7 it follows that if $j < k \leq n$, then it must be the case that $ind(et_j) \geq ind(et_k)$.

We henceforth consider only well formed expressions and this qualification is assumed implicitly whenever we speak of terms, environments, environment terms or expressions.

4.2 Rules for rewriting expressions

As explained earlier, suspensions are introduced into our notation for terms so as to permit a laziness in the substitution operation needed in β -contraction. Accordingly, β -contraction will now be realized through a rule that generates a suspension and a series of rules that permit the suspension to be ‘propagated’ over the structure of a term. In the course of such a propagation, it

- (m1) $\llbracket [t, ol_1, nl_1, e_1], ol_2, nl_2, e_2 \rrbracket \rightarrow \llbracket t, ol', nl', \{e_1, nl_1, ol_2, e_2\} \rrbracket$,
 where $ol' = ol_1 + (ol_2 \dot{-} nl_1)$ and $nl' = nl_2 + (nl_1 \dot{-} ol_2)$.
- (m2) $\{\{nil, nl, 0, nil\}\} \rightarrow nil$.
- (m3) $\{\{nil, nl, ol, et :: e\}\} \rightarrow \{\{nil, nl', ol', e\}\}$,
 where $nl, ol \geq 1$, $nl' = nl - 1$ and $ol' = ol - 1$.
- (m4) $\{\{nil, 0, ol, e\}\} \rightarrow e$.
- (m5) $\{\{et :: e_1, nl, ol, e_2\}\} \rightarrow \langle\langle et, nl, ol, e_2 \rangle\rangle :: \{\{e_1, nl, ol, e_2\}\}$.
- (m6) $\langle\langle et, nl, 0, nil \rangle\rangle \rightarrow et$.
- (m7) $\langle\langle @n, nl, ol, @l :: e \rangle\rangle \rightarrow @m$,
 where $m = l + (nl \dot{-} ol)$, provided $nl = n + 1$.
- (m8) $\langle\langle @n, nl, ol, (t, l) :: e \rangle\rangle \rightarrow (t, m)$,
 where $m = l + (nl \dot{-} ol)$, provided $nl = n + 1$.
- (m9) $\langle\langle (t, nl), nl, ol, et :: e \rangle\rangle \rightarrow (\llbracket t, ol, l', et :: e \rrbracket, m)$
 where $l' = ind(et)$ and $m = l' + (nl \dot{-} ol)$.
- (m10) $\langle\langle et, nl, ol, et' :: e \rangle\rangle \rightarrow \langle\langle et, nl', ol', e \rangle\rangle$,
 where $nl' = nl - 1$ and $ol' = ol - 1$, provided $nl \neq ind(et)$.

Figure 4: Rule schemata for merging suspensions

is possible for two suspensions to ‘meet’ each other. Our system includes rules for ‘merging’ such suspensions, thus permitting them to be propagated in a combined walk over the structure of the embedded term.

Our rules are presented through schemata divided into three categories: the β_s -*contraction* rule schema, the *reading* rule schemata and the *merging* rule schemata. Each category is presented separately in Figures 2–4. The following tokens, used in these schemata perhaps with subscripts or superscripts, are to be interpreted as schema variables for the indicated syntactic categories: c for constants, t for terms, et for environment terms, e for environments, i and j for positive numbers and ol , nl , l , m and n for natural numbers. The applicability of several of the rule schemata are dependent on ‘side’ conditions that are presented together with them. Further, in determining the relevant instance of the righthand side of some of the rule schemata, simple arithmetic operations may have to be performed on components of the expression matching the lefthand side. In the discussions that follow, we shall often include these arithmetic operations within the expression being written. Using this convention, rule (m1) in Figure 4 may also be written as

$$\llbracket [t_1, ol_1, nl_1, e_1], ol_2, nl_2, e_2 \rrbracket \rightarrow \llbracket t, ol_1 + (ol_2 \dot{-} nl_1), nl_2 + (nl_1 \dot{-} ol_2), \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket.$$

Given the syntax of expressions, this convention is really an abuse of notation. However, this abuse is harmless and unambiguous and is, in addition, extremely convenient.

Definition 4.8 The reduction relations generated by the rule schemata in Figure 2, 3 and 4 are denoted by \triangleright_{β_s} , \triangleright_r , \triangleright_m respectively. The union of the relations \triangleright_r and \triangleright_m is denoted by \triangleright_{rm} , the union of \triangleright_r and \triangleright_{β_s} by $\triangleright_{r\beta_s}$ and the union of \triangleright_r , \triangleright_m and \triangleright_{β_s} by $\triangleright_{rm\beta_s}$.

The legitimacy of the above definitions is dependent on the rewrite rules presented above producing well formed expressions from well formed expressions. The following sequence of observations culminating in Theorem 4.12 establishes this fact.

Lemma 4.9 *If e_1 is an environment and $e_1 \triangleright_{rm\beta_s} e_2$ then $len(e_1) = len(e_2)$.*

Proof. Let e_1 be an environment. Then the following fact is easily established by induction on the structure of e_1 : if x_1 is a subexpression of e_1 and x_2 is an expression of the same type as x_1 such that $len(x_1) = len(x_2)$ in the case that x_1 is an environment, and if e_2 is obtained from e_1 by replacing x_1 by x_2 , then $len(e_1) = len(e_2)$. The desired conclusion would then follow if whenever x_1 is an environment and $x_1 \rightarrow x_2$ is an instance of one of the rule schemata in Figures 2–4, then $len(x_1) = len(x_2)$. This can be seen to be the case by inspecting the relevant schemata, namely (m2), (m3), (m4) and (m5). □

Lemma 4.10 *Let $x_1 \rightarrow x_2$ be an instance of some schema in Figures 2–4. If x_1 is an environment term then $ind(x_1) = ind(x_2)$. If x_1 is an environment, then, for every natural number l , $ind_l(x_1) = ind_l(x_2)$.*

Proof. By a routine, if somewhat tedious, inspection of the relevant rule schemata, namely (m2)–(m10). □

Lemma 4.11 *If x_1 is an environment term or an environment and $x_1 \triangleright_{rm\beta_s} x_2$, then $ind(x_1) = ind(x_2)$.*

Proof. Let x_1 and x_2 both be environment terms or environments with the following property: if x_1 is an environment term then $ind(x_1) = ind(x_2)$ and if x_1 is an environment then, for every natural number l , $ind_l(x_1) = ind_l(x_2)$. The following facts are easily established by a simultaneous induction on the structure of expressions: If y_1 is an environment term with x_1 as a subexpression and y_2 results from y_1 by replacing x_1 by x_2 , then $ind(y_1) = ind(y_2)$. If y_1 is an environment instead and y_2 results from it by a similar replacement, then, for every natural number l , $ind_l(y_1) = ind_l(y_2)$. The desired conclusion now follows easily from Lemma 4.10. \square

Theorem 4.12 *Let x be a well formed expression and let y be such that $x \triangleright_r y$, $x \triangleright_m y$, $x \triangleright_{\beta_s} y$, $x \triangleright_{rm} y$, $x \triangleright_{r\beta_s} y$ or $x \triangleright_{rm\beta_s} y$. Then y is a well formed expression.*

Proof. It is sufficient to show that this property holds if $x \triangleright_{rm\beta_s} y$. Given Lemmas 4.9 and 4.11, this would be true if whenever x_1 is a well formed expression and $x_1 \rightarrow x_2$ is an instance of some schema in Figures 2–4, then x_2 is well formed. This is verified by an inspection of the relevant schemata. The argument is routine in all cases except those of schemata (m1) and (m5). In the case of (m1), *i.e.*, when the rule is

$$\llbracket [t_1, ol_1, nl_1, e_1], ol_2, nl_2, e_2 \rrbracket \rightarrow \llbracket t, ol_1 + (ol_2 \div nl_1), nl_2 + (nl_1 \div ol_2), \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket,$$

some care is needed in verifying that $ind(\{\{e_1, nl_1, ol_2, e_2\}\}) \leq nl_2 + (nl_1 \div ol_2)$. In the case when $len(e_1) = 0$ or $len(e_1) > 0$ and $len(e_2) > (nl_1 - ind_0(e_1))$, this follows from the fact that $ind_l(e_2) \leq nl_2$. In the only remaining case, $ind_0(e_1) \leq (nl_1 - len(e_2))$. Noting that in this case $ind(\{\{e_1, nl_1, ol_2, e_2\}\}) = ind_0(e_1)$ and that $len(e_2) = ol_2$, the desired conclusion is obtained. In the case of (m5), *i.e.*, when the rule is

$$\{\{et :: e_1, j, k, e_2\}\} \rightarrow \langle\langle et, j, k, e_2 \rangle\rangle :: \{\{e_1, j, k, e_2\}\}.$$

we need to verify that $ind(\langle\langle et, j, k, e_2 \rangle\rangle) \geq ind(\{\{e_1, j, k, e_2\}\})$. However, this is done easily using Lemmas 4.10 and 4.7. \square

We illustrate the rewrite rules presented in this section by considering their use on the term $((\lambda((\lambda(\lambda((\#1 \#2) \#3))) t_2)) t_3)$, assuming that t_2 and t_3 are arbitrary de Bruijn terms. The following constitutes a $\triangleright_{rm\beta_s}$ -reduction sequence on this term:

$$\begin{aligned} & ((\lambda((\lambda(\lambda((\#1 \#2) \#3))) t_2)) t_3) \\ & \triangleright_{\beta_s} \llbracket ((\lambda(\lambda((\#1 \#2) \#3))) t_2), 1, 0, (t_3, 0) :: nil \rrbracket \\ & \triangleright_{\beta_s} \llbracket \llbracket (\lambda((\#1 \#2) \#3)), 1, 0, (t_2, 0) :: nil \rrbracket, 1, 0, (t_3, 0) :: nil \rrbracket \\ & \triangleright_m \llbracket (\lambda((\#1 \#2) \#3)), 2, 0, \{\{(t_2, 0) :: nil, 0, 1, (t_3, 0) :: nil\}\} \rrbracket \\ & \triangleright_m \llbracket (\lambda((\#1 \#2) \#3)), 2, 0, \langle\langle (t_2, 0), 0, 1, (t_3, 0) :: nil \rangle\rangle :: \{\{nil, 0, 1, (t_3, 0) :: nil\}\} \rrbracket \\ & \triangleright_m \llbracket (\lambda((\#1 \#2) \#3)), 2, 0, (\llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket, 0) :: \{\{nil, 0, 1, (t_3, 0) :: nil\}\} \rrbracket \\ & \triangleright_m \llbracket (\lambda((\#1 \#2) \#3)), 2, 0, (\llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket, 0) :: (t_3, 0) :: nil \rrbracket. \end{aligned}$$

The combined environment can then be moved inside the abstraction by using a reading rule to yield the term

$$(\lambda \llbracket (\#1 \#2) \#3 \rrbracket, 3, 1, @0 :: (\llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket, 0) :: (t_3, 0) :: nil \rrbracket).$$

A repeated application of reading rules transforms the last term into

$$(\lambda ((\#1 \llbracket \llbracket t_2, 1, 0, (t_3, 0) :: nil \rrbracket, 0, 1, nil \rrbracket \rrbracket \llbracket t_3, 0, 1, nil \rrbracket)).$$

Applying merging rules to this term produces the term

$$(\lambda ((\#1 \llbracket t_2, 1, 1, (t_3, 0) :: nil \rrbracket \rrbracket \llbracket t_3, 0, 1, nil \rrbracket)).$$

Depending on the particular structures of t_2 and t_3 , the reading rules can be applied repeatedly to this term to finally produce a de Bruijn term that results from the original term by contracting the two outermost β -redexes. A noteworthy point in connection with this example is the manner in which the ability to perform substitutions lazily and to merge environments enables ‘substitution walks’ to be combined.

5 A well founded partial order on suspension expressions

We will need, in later discussions, to compare suspension expressions. To facilitate such comparisons, we now define a well founded partial ordering relation between such expressions. Intrinsic to this relation are the measures on expressions contained in the following definition. We assume below that \max represents the function on pairs of integers that picks the larger integer in the pair.

Definition 5.1 The measures η on expressions and μ on terms are defined recursively in the table below:

exp	$\eta(exp)$	$\mu(exp)$
constant	0	1
$\#i$	0	1
$(t_1 t_2)$	$\max(\eta(t_1), \eta(t_2))$	$\max(\mu(t_1), \mu(t_2)) + 1$
(λt)	$\eta(t)$	$\mu(t) + 1$
$\llbracket t, ol, nl, e \rrbracket$	$\mu(t) + \eta(e)$	$\mu(t) + \eta(e) + 1$
nil	0	—
$et :: e$	$\max(\eta(et), \eta(e))$	—
$\{\{e_1, nl, ol, e_2\}\}$	$\eta(e_1) + \eta(e_2) + 1$	—
$@l$	0	—
(t, l)	$\mu(t)$	—
$\langle\langle et, nl, ol, e \rangle\rangle$	$\eta(et) + \eta(e) + 1$	—

We observe the following properties of the measures just defined.

Lemma 5.2 *For any expression x , $\eta(x) \geq 0$. Further, for any term t , $\mu(t) > \eta(t)$.*

Proof. Obvious induction on the structure of expressions and terms. □

Lemma 5.3 *Let x_1 and x_2 be expressions of the same syntactic category and such that $\eta(x_1) \geq \eta(x_2)$ and, if x_1 and x_2 are terms, $\mu(x_1) \geq \mu(x_2)$. If x results from y by the replacement of subexpression x_1 by x_2 , then $\eta(y) \geq \eta(x)$ and, if x and y are terms, $\mu(y) \geq \mu(x)$.*

Proof. Induction on the structure of expressions and an inspection of Definition 5.1. □

Our ordering relation on expressions has the following character: It first compares the ‘complexities’ of two expressions directly. If it cannot distinguish between them at this level, it compares, recursively, their subexpressions. The following notion is needed in stating this idea precisely.

Definition 5.4 Two expressions are said to have the same top-level structure if they are both constants, variable references, abstractions, applications, or suspensions or if they are both of the forms nil , $et :: e$, $\{\{e_1, i, j, e_2\}\}$, $@l$, (t, l) , or $\langle\langle et, i, j, e \rangle\rangle$. If two suspension expressions that have the same top-level structure have any immediate subexpressions, then there is an obvious correspondence between these subexpressions. This correspondence will be utilized below.

We now define a relation for comparing expressions.

Definition 5.5 Given two expressions x_1 and x_2 , we say $x_1 \sqsupseteq x_2$ if either $\eta(x_1) > \eta(x_2)$ or $\eta(x_1) = \eta(x_2)$ and one of the following conditions hold:

- (1) x_1 is $\#i$ and x_2 is $\#j$ where $i > j$.
- (2) x_1 is $\llbracket t_1, ol_1, nl_1, e_1 \rrbracket$, x_2 is $\llbracket t_2, ol_2, nl_2, e_2 \rrbracket$ and $\eta(t_1) > \eta(t_2)$.
- (3) x_1 is $\{\{e_1, nl, ol, e_2\}\}$, x_2 is $et :: e$ and $x_1 \sqsupseteq e$.
- (4) x_1 and x_2 have the same top-level structure and also have immediate subexpressions such that each immediate subexpression of x_1 is identical to the corresponding immediate subexpression of x_2 except for one pair of immediate subexpressions x'_1 of x_1 and x'_2 of x_2 for which $x'_1 \sqsupseteq x'_2$.
- (5) x_2 is an immediate subexpression of x_1 .

We shall write $x_1 \sqsupseteq x_2$ to signify that $x_1 = x_2$ or $x_1 \sqsupseteq x_2$.

The relation defined above is not a partial ordering relation because it is not transitive.⁴ However, its transitive closure will yield a well founded partial ordering relation. The following lemma is useful in showing that this is indeed the case.

Lemma 5.6 *There is no infinite sequences of expressions $x_1, x_2, \dots, x_n, \dots$ such that*

⁴A partial ordering relation has been described in the literature to be one that is irreflexive and transitive [Lev79] as well as to be one that is reflexive, transitive and antisymmetric [Hal60]. It is the former definition that we use here.

$$x_1 \sqsupset x_2 \sqsupset \cdots \sqsupset x_n \sqsupset \cdots.$$

Proof. Let us use the phrase “infinite descending sequence” to denote an infinite sequence of expressions $x_1, x_2, \dots, x_n, \dots$ such that

$$x_1 \sqsupset x_2 \sqsupset \cdots \sqsupset x_n \sqsupset \cdots.$$

We then prove the following by induction on $\eta(x_1)$:

- (a) There is no infinite descending sequence of expressions $x_1, x_2, \dots, x_n, \dots$ such that $\eta(x_1) = \eta(x_2) = \cdots = \eta(x_n) = \cdots$.
- (b) There is no infinite descending sequence of expressions.

We note first that if $x \sqsupset y$, then $\eta(x) \geq \eta(y)$. From this we see that (b) is a consequence of (a) and the hypothesis. Thus, it is only necessary to show (a). We do this by an induction on the structure of x . The argument proceeds by considering the various possibilities for this structure.

If x is a term: If x is a constant, then x is minimal with respect to \sqsupset and so the claim must be true. If x is of the form $\#k$, any descending chain can be of length at most $(k - 1)$. Suppose x is an application. Let $x = (s_1 t_1)$. Now, any infinite descending sequence of expressions starting at x and preserving the η values of expressions must be of one of two forms:

$$(s_1 t_1), (s_2 t_2), \dots, (s_n t_n), \dots,$$

where, for $i \geq 1$, either $s_i = s_{i+1}$ and $t_i \sqsupset t_{i+1}$ or $s_i \sqsupset s_{i+1}$ and $t_i = t_{i+1}$, or

$$(s_1 t_1), (s_2 t_2), \dots, (s_n t_n), r_{n+1}, \dots,$$

where, for $1 \leq i < n$, either $s_i = s_{i+1}$ and $t_i \sqsupset t_{i+1}$ or $s_i \sqsupset s_{i+1}$ and $t_i = t_{i+1}$ and r_{n+1} is either s_n or t_n . In either case, it is easily seen that there must then be an infinite descending sequence that starts at either s_1 or t_1 . However, $\eta(s_1), \eta(t_1) \leq \eta(x)$ and s_1 and t_1 are subexpressions of x , and so this is impossible by hypothesis.

An argument similar to that in the case of an application can be provided when x is an abstraction. This leaves only the case of a suspension. Let $x = \llbracket s_1, ol_1, nl_1, e_1 \rrbracket$. In this case, we use an additional induction on $\eta(s_1)$. By virtue of Lemma 5.2, $\eta(x) > \eta(s_1)$ and $\eta(x) > \eta(e_1)$. There are, therefore, no infinite descending sequences from s_1 or e_1 . Using this and an argument similar to that in the case of an application, we see that a purportedly infinite descending sequence starting from t must have an initial segment of the form

$$\llbracket s_1, ol_1, nl_1, e_1 \rrbracket, \llbracket s_2, ol_2, nl_2, e_2 \rrbracket, \dots, \llbracket s_n, ol_n, nl_n, e_n \rrbracket, \llbracket s_{n+1}, ol_{n+1}, nl_{n+1}, e_{n+1} \rrbracket,$$

where, for $1 \leq i < n$, $s_i \sqsupseteq s_{i+1}$ and $e_i \sqsupseteq e_{i+1}$, and $\eta(s_n) > \eta(s_{n+1})$. Now, clearly, $\eta(s_1) > \eta(s_{n+1})$ and, by assumption, $\eta(x) = \eta(\llbracket s_{n+1}, ol_{n+1}, nl_{n+1}, e_{n+1} \rrbracket)$. Thus, such an initial segment cannot exist if $\eta(s_1) = 0$. Furthermore, even if $\eta(s_1) > 0$, the segment cannot be extended into an infinite descending sequence: that would entail the existence of an infinite descending sequence

from $\llbracket s_{n+1}, ol_{n+1}, nl_{n+1}, e_{n+1} \rrbracket$, in contradiction to the hypothesis. The claim must, therefore, be true in this case as well.

If x is an environment term: If x is of the form $@l$, it is minimal and so the claim follows. If it is of the form (t', l) , there is an infinite descending sequence from x only if there is one from t' . However, using Lemma 5.2, $\eta(x) > \eta(t')$. Thus, this is impossible by hypothesis. Finally, consider the case when x is of the form $\langle\langle et, nl, ol, e \rangle\rangle$. There can be an infinite descending sequence from x only if there is one from either et or e . This is, again, impossible by hypothesis, because $\eta(x) > \eta(et)$ and $\eta(x) > \eta(e)$.

If x is an environment: If x is nil , it is a minimal expression, and so the claim clearly holds. Let $x = et :: e$. Using an argument similar to that in the case of an application, we see that there is an infinite descending sequence starting at x only if there is also one starting at et or e . However this is impossible by hypothesis, because $\eta(x) \geq \eta(et)$, $\eta(x) \geq \eta(e)$, and et and e are subexpressions of x .

The remaining case, where x is of the form $\{\{e_1, nl, ol, e'_1\}\}$, requires a non-constructive proof. Let us assume that there are infinite descending sequences starting at x . We pick from these a sequence $x = y_1, y_2, y_3, \dots$ that is minimal in the following sense: for each $i \geq 1$, there is no infinite descending sequence of the form $y_1, y_2, \dots, y_i, y'_{i+1}, \dots$ where y'_{i+1} is a subexpression of y_{i+1} . We focus now on the sequence picked. We observe first that $\eta(x) > \eta(e_1)$ and $\eta(x) > \eta(e'_1)$. Thus, by hypothesis, there are no infinite descending sequences starting at either e_1 or e'_1 . From this, it is easily seen that our sequence must be of the form

$$\{\{e_1, nl, ol, e'_1\}\}, \{\{e_2, nl, ol, e'_2\}\}, \dots, \{\{e_n, nl, ol, e'_n\}\}, et_{n+1} :: e_{n+1}, \dots,$$

where, for $1 \leq i < n$, $e_i \sqsupseteq e_{i+1}$, $e'_i \sqsupseteq e'_{i+1}$ and $\{\{e_n, nl, ol, e'_n\}\} \sqsupset e_{n+1}$. Now, this infinite descending sequence entails that there is a similar sequence starting from $et_{n+1} :: e_{n+1}$. Using an argument whose structure is by now familiar, this can be the case only if there is an infinite descending sequence z_1, z_2, z_3, \dots , where z_1 is either et_{n+1} or e_{n+1} . We note that $\eta(et_{n+1}) \leq \eta(x)$ and that et_{n+1} is an environment term. Thus, we have already shown that the former situation is impossible. In the latter case, we can construct the infinite descending sequence

$$\{\{e_1, nl, ol, e'_1\}\}, \{\{e_2, nl, ol, e'_2\}\}, \dots, \{\{e_n, nl, ol, e'_n\}\}, z_1, z_2, z_3, \dots$$

However, this contradicts our assumption of minimality for the sequence picked initially. We conclude, therefore, that no infinite sequence could have existed to begin with.

All the cases having been considered, our claim stands verified and so the lemma must be true. □

We now define the desired ordering relation on expressions.

Definition 5.7 The relation \succ on expressions is the transitive closure of the relation \sqsupset .

Theorem 5.8 *The relation \succ is a well founded partial ordering relation on expressions.*

Proof. We need to show that \succ is irreflexive and, assuming that it is a partial order, is also well founded. Both requirements follow from the observation that there can be no infinite descending chains relative to \succ , a fact that is an obvious consequence of Lemma 5.6. □

The proof we have provided for the fact that \succ is well founded is a direct one and has the virtue of giving us specific insight into the nature of this relation. However, an alternative proof can be provided by invoking Kruskal's tree theorem [Gal91, Kru60], thereby exhibiting relationships between \succ and the notions of simplification orderings [Der82] and Kamin and Lévy's extended recursive path orderings (described, for example, in [Hue86]). Towards this end, we note that expressions of the form $\llbracket t, ol, nl, e \rrbracket$, $\{\{e_1, nl, ol, e_2\}\}$ and $\langle\langle et, nl, ol, e \rangle\rangle$ can be thought of as functions of two arguments by incorporating nl and ol into the name of the function symbol; thus, the first expression may be rendered into the expression $f_{ol,nl}(t, e)$, the second into $g_{nl,ol}(e_1, e_2)$ and the third into $h_{nl,ol}(et, e)$. In a similar fashion, an environment term of the form (t, l) could be rendered into the expression $k_l(t)$, *i.e.*, a function of one argument. Finally, expressions of the form $(t_1 t_2)$ and (λt) can be translated into $app(t_1, t_2)$ and $lam(t)$, respectively, $::$ can be interpreted as a binary function symbol and expressions of the form $\#k$, $@l$ and nil can be thought of as constants. Given such a translation, \succ can be seen to be a simplification ordering. This alone does not allow us to conclude that \succ is well founded, since the alphabet over which our terms are constructed is infinite. However, let \approx be the relation over this alphabet that includes the identity relation and is such that (i) $f_{ol,nl} \approx f_{ol',nl'}$, $g_{nl,ol} \approx g_{nl',ol'}$ and $h_{nl,ol} \approx h_{nl',ol'}$ for all ol, ol', nl, nl' , (ii) $k_l \approx k_{l'}$ and $@l \approx @l'$ for all l, l' , (iii) $\#i \approx \#j$ if $i \leq j$, and (iv) $c \approx c'$ for all constants c and c' of the original vocabulary. It is easily seen that \approx is a well quasi ordering relation on the alphabet. Now, let \preceq be the homeomorphic embedding of \approx . By Kruskal's theorem, \preceq is a well quasi order on expressions. We observe at this point that if $x \succ y$, then it cannot be the case that $x \preceq y$. From this it follows that \succ is well founded.

6 The meaning of the reading and merging rules

The purpose of the merging and reading rules is that of propagating substitutions embodied in suspension expressions. To be meaningful from this perspective, these rules must be capable of eventually transforming any given expression in our notation into ones that are 'substitution-free'. Thus, when restricted to terms, a repeated use of these rules must produce a de Bruijn term from any suspension term. Furthermore, the expression that is eventually produced from a given expression should be independent of the order in which rules are applied and, in fact, even the choice of rules. Recast in the terminology of rewrite systems, these requirements amount to the following: (i) every suspension expression should have a unique \triangleright_{rm} -normal form and (ii) the \triangleright_{rm} -normal form of a suspension term should be a de Bruijn term.

We show, in this section, that these requirements hold of the reading and merging rules. We begin by observing a generalization of the second requirement. As a consequence of the theorem below, we also see that an expression in \triangleright_{rm} -normal form is a simple expression.

Theorem 6.1 *An expression x is in \triangleright_{rm} -normal form if and only if one of the following holds: (a) x is a de Bruijn term; (b) x is an environment term of the form $@l$ or (t, l) where t is a term in \triangleright_{rm} -normal form; or (c) x is an environment of the form nil or $et :: e$ where et and e are, respectively, an environment term and an environment in \triangleright_{rm} -normal form.*

Proof. An inspection of Figures 3 and 4 shows that a well formed expression that has a subexpression of the form $\llbracket t, i, j, e \rrbracket$, $\{\{e_1, i, j, e_2\}\}$ or $\langle\langle et, i, j, e \rangle\rangle$ can be rewritten by using one of the rule schemata appearing in these figures. Thus, such an expression cannot be in \triangleright_{rm} -normal form. \square

We wish to show that a \triangleright_{rm} -normal form exists for every expression. Actually a stronger result holds: the relation \triangleright_{rm} is *noetherian*. We prove this by showing that each application of a merging or reading rule decreases the complexity of an expression in the sense of the well founded partial ordering relation defined in Section 5.

Lemma 6.2 *If $l \rightarrow r$ is an instance of one of the rule schemata in Figures 3 or 4, then $\eta(l) \geq \eta(r)$ and, if l and r are terms, $\mu(l) \geq \mu(r)$.*

Proof. By a routine, but tedious, inspection of the rules in question. We omit the details, but note that in all cases except when the rule is an instance of (r5), (m1), (m3), (m5) or (m10), $\eta(l) > \eta(r)$. \square

Lemma 6.3 *If x_1 and x_2 are expressions such that $x_1 \triangleright_{rm} x_2$, then $\eta(x_1) \geq \eta(x_2)$.*

Proof. This follows immediately from Lemmas 5.3 and 6.2. \square

Lemma 6.4 *If $l \rightarrow r$ is an instance of one of the rule schemata in Figures 3 or 4, then $l \succ r$.*

Proof. Immediate from the fact noted in the proof of Lemma 6.2 in all cases except when the rule is an instance of (r5), (m1), (m3), (m5) or (m10). In the cases left, the lemma is easily shown to be true using Lemma 6.2 and inspecting Definitions 5.5 and 5.7. It is necessary only to note, for (m1), that $\eta(\llbracket t, ol, nl, e \rrbracket) \geq \mu(t)$ and, by Lemma 5.2, $\mu(t) > \eta(t)$. \square

Lemma 6.5 *If $x_1 \triangleright_{rm} x_2$, then $x_1 \succ x_2$.*

Proof. By induction on the structure of x_1 . If $x_1 \rightarrow x_2$ is an instance of one of the rule schemata in Figures 3 or 4, this follows from Lemma 6.4. Otherwise x_1 and x_2 have the same top-level structure and, by Lemma 6.3, $\eta(x_1) \geq \eta(x_2)$. If $\eta(x_1) > \eta(x_2)$, the desired conclusion follows. If $\eta(x_1) = \eta(x_2)$, by the definition of \triangleright_{rm} and by the hypothesis, there is an immediate subexpression x'_1 of x_1 and a corresponding immediate subexpression x'_2 of x_2 such that $x'_1 \succ x'_2$ and every other immediate subexpressions of x_1 is identical to the corresponding immediate subexpression of x_2 . Using the definition of \succ , it follows easily that $x_1 \succ x_2$. \square

Theorem 6.6 *The relation \triangleright_{rm} is noetherian.*

Proof. An obvious consequence of Lemma 6.5 and Theorem 5.8. □

We have thus seen that a \triangleright_{rm} -normal form exists for every expression and we have also observed the shape of such a form. We now wish to show the uniqueness of \triangleright_{rm} -normal forms. By virtue of Proposition 2.1, these forms would be unique if \triangleright_{rm} is a confluent reduction relation. In light of Proposition 2.2 and Theorem 6.6 it is actually enough to show that \triangleright_{rm} is locally confluent. We establish the local confluence of \triangleright_{rm} through a series of observations, culminating in Theorem 6.17. We begin with an observation about arithmetic on natural numbers that is useful in arguing the identity of indices of environment terms.

Lemma 6.7 $((i + (j \dot{-} k)) \dot{-} l) = (i \dot{-} l) + (j \dot{-} (k + (l \dot{-} i)))$.

Proof. $((i + (j \dot{-} k)) \dot{-} l) = (i \dot{-} l) + ((j \dot{-} k) \dot{-} (l \dot{-} i)) = (i \dot{-} l) + (j \dot{-} (k + (l \dot{-} i)))$. □

We now observe some reduction properties of expressions in special forms. These observations are used eventually in Theorem 6.17.

Lemma 6.8 *Let e be a simple environment. Then*

$$\llbracket \#i, ol, nl, e \rrbracket \triangleright_{rm}^* \begin{cases} \#(i + (nl - ol)) & \text{if } i > ol \\ \#(nl - m) & \text{if } i \leq ol \text{ and } e[i] = @m \\ \llbracket t, 0, nl - m, nil \rrbracket & \text{if } i \leq ol \text{ and } e[i] = (t, m). \end{cases}$$

Proof. By an induction on ol if $i > ol$ and on i if $i \leq ol$. The rule schemata (r2)–(r5) are used in an obvious way in this proof. □

Lemma 6.9 *Let e be a simple environment. If $(nl - l) \geq ol$, then $\llbracket (t, l), nl, ol, e \rrbracket \triangleright_{rm}^* \llbracket t, l \rrbracket$. If $(nl - l) < ol$, then $\llbracket (t, l), nl, ol, e \rrbracket \triangleright_{rm}$ -reduces to*

$$\llbracket \llbracket t, ol - (nl - l), ind(e[nl - l + 1]), e\{nl - l + 1\} \rrbracket, ind(e[nl - l + 1]) + (nl \dot{-} ol) \rrbracket.$$

Proof. If $(nl - l) < ol$, we use an induction on $(nl - l)$ and if $(nl - l) \geq ol$, we use an induction on ol . Rule schemata (m10), (m9) and (m6) are used in this proof. □

Lemma 6.10 *Let e be a simple environment. Then*

$$\llbracket @l, nl, ol, e \rrbracket \triangleright_{rm}^* \begin{cases} @l & \text{if } (nl - l) > ol \\ @m + (nl \dot{-} ol) & \text{if } (nl - l) \leq ol \text{ and } e[nl - l] = @m \\ (t, m + (nl \dot{-} ol)) & \text{if } (nl - l) \leq ol \text{ and } e[nl - l] = (t, m). \end{cases}$$

Proof. Analogous to that of Lemma 6.9, using rule schemata (m6)–(m8) and (m10). □

Lemma 6.11 *Let et be an environment term such that $ind(et) \leq nl$. Then, for $j \geq 1$,*

$$\langle\langle et, nl + j, ol + j, et_1 :: \dots :: et_j :: e \rangle\rangle \triangleright_{rm}^* \langle\langle et, nl, ol, e \rangle\rangle.$$

Proof. By an induction on j , using rule schema (m10). □

Lemma 6.12 *Let e_2 be a simple environment. Then*

$$\{\{nil, nl, ol, e_2\}\} \triangleright_{rm}^* \begin{cases} nil & \text{if } nl \geq ol \\ e_2\{nl + 1\} & \text{otherwise.} \end{cases}$$

Proof. By an induction on ol if $nl \geq ol$ and on nl if $nl < ol$. Rule schemata (m2)–(m4) are used in this proof. □

Lemma 6.13 *Let e_1 be a simple environment and let nl and ol be natural numbers such that $(nl - ind(e_1)) \geq ol$. Then $\{\{e_1, nl, ol, e_2\}\} \triangleright_{rm}^* e_1$.*

Proof. We may assume that e_2 is also a simple environment. If not, it can be \triangleright_{rm} -reduced to one and then the argument given here can be used. We now use an induction on $len(e_1)$. If this is 0, then $e_1 = nil$. Since $ind(nil) = 0$, $nl \geq ol$ and so, by Lemma 6.12, $\{\{e_1, nl, ol, e_2\}\} \triangleright_{rm}^* nil$. If $len(e_1) > 0$, e_1 is of the form $et_1 :: e'_1$. Using rule schema (m5),

$$\{\{e_1, nl, ol, e_2\}\} \triangleright_{rm}^* \langle\langle et_1, nl, ol, e_2 \rangle\rangle :: \{\{e'_1, nl, ol, e_2\}\}.$$

We note that $ind(et_1) = ind(e_1)$ and, by the definition of wellformedness, $ind(e'_1) \leq ind(e_1)$. The lemma then follows from Lemma 6.11, rule schema (m6) and the inductive hypothesis. □

Lemma 6.14 *If e_1 is an environment such that $ind(e_1) \leq nl$ then, for $j \geq 1$, the expressions*

$$\{\{e_1, nl + j, ol + j, et_1 :: \dots :: et_j :: e_2\}\} \quad \text{and} \quad \{\{e_1, nl, ol, e_2\}\}$$

\triangleright_{rm} -reduce to a common expression for any environment e_2 .

Proof. We may assume that e_1 and e_2 are simple expressions: if they are not already in this form, they can be \triangleright_{rm} -reduced to expressions that are in such a form in both given expressions and, since, by Lemma 4.11, $ind(e_1)$ is preserved by such a reduction, we can then invoke the argument provided here. We now use an induction on $len(e_1)$. If this is 0, *i.e.*, if $e_1 = nil$, using Lemma 6.12 we see that both expressions \triangleright_{rm} -reduce to nil if $nl \geq ol$ and to $e_2\{nl + 1\}$ otherwise. If $len(e_1) > 0$, then e_1 is of the form $et' :: e'_1$. In this case, by using rule schema (m5),

$$\{\{e_1, nl + j, ol + j, et_1 :: \dots :: et_j :: e_2\}\} \triangleright_{rm} \langle\langle et', nl + j, ol + j, et_1 :: \dots :: et_j :: e_2 \rangle\rangle :: \{\{e'_1, nl + j, ol + j, et_1 :: \dots :: et_j :: e_2\}\},$$

and, similarly,

$$\{\{e_1, nl, ol, e_2\}\} \triangleright_{rm} \langle\langle et', nl, ol, e_2 \rangle\rangle :: \{\{e'_1, nl, ol, e_2\}\}.$$

Noting that $ind(et') = ind(e_1)$ and $ind(e'_1) \leq ind(e_1)$, Lemma 6.11 and the inductive hypothesis yield the desired conclusion. \square

Lemma 6.15 *Let a and b be environment terms of the form*

$$\langle\langle\langle\langle et_1, nl_1, ol_2, e_2 \rangle\rangle, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \rangle\rangle \quad \text{and} \\ \langle\langle et_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{\{e_2, nl_2, ol_3, e_3\}\}\rangle\rangle,$$

respectively. Then there is an environment term r such that $a \triangleright_{rm}^ r$ and $b \triangleright_{rm}^* r$.*

Proof. We assume that et_1 , e_2 and e_3 are simple expressions. If this is not the case, they can be \triangleright_{rm} -reduced to simple expressions in both a and b and the argument that we provide here can then be applied.

We now prove the lemma by an induction on $len(e_2)$.

Base Case: $len(e_2) = 0$. In this case, since $ol_2 = 0$ and $e_2 = nil$, a and b are, in fact, the environment terms $\langle\langle\langle\langle et_1, nl_1, 0, nil \rangle\rangle, nl_2 + nl_1, ol_3, e_3 \rangle\rangle$ and $\langle\langle et_1, nl_1, ol_3 \dot{-} nl_2, \{\{nil, nl_2, ol_3, e_3\}\}\rangle\rangle$, respectively. Using rule schema (m6), it follows that

$$a \triangleright_{rm}^* \langle\langle et_1, nl_2 + nl_1, ol_3, e_3 \rangle\rangle. \tag{1}$$

Our analysis now splits into two subcases, depending on whether or not $nl_2 \geq ol_3$. If $nl_2 \geq ol_3$, then, using Lemma 6.12, it can be seen that $b \triangleright_{rm}^* \langle\langle et_1, nl_1, 0, nil \rangle\rangle$. Noting that $ind(et_1) \leq nl_1$ and using Lemma 6.9 or 6.10 depending on the form of et_1 on the one hand and using rule schema (m6) on the other, we see that a and b both \triangleright_{rm} -reduce to et_1 . In the other case, *i.e.*, when $nl_2 < ol_3$, using Lemma 6.12, we observe that $b \triangleright_{rm}$ -reduces to $\langle\langle et_1, nl_1, ol_3 \dot{-} nl_2, e_3\{nl_2 + 1\}\rangle\rangle$. Using Lemma 6.11 in conjunction with (1), we see that a also \triangleright_{rm} -reduces to the same expression.

Inductive Step: $len(e_2) > 0$. Let e_2 be of the form $et_2 :: e'_2$. We now use a further induction on $nl_1 - ind(et_1)$.

Base Case for Second Induction: $nl_1 - ind(et_1)$ is 0. We consider the cases for the structure of et_1 :

(a) et_1 is of the form $@l$. We note first that $l = nl_1 - 1$. Now, our analysis splits into two further subcases, depending on whether et_2 is of the form $@m$ or of the form (t, m) .

In the former situation, using Lemma 6.10 on the one hand and rule schema (m5) on the other, it can be seen that

$$a \triangleright_{rm}^* \langle\langle @ (m + (nl_1 \dot{-} ol_2)), nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \rangle\rangle \quad \text{and} \\ b \triangleright_{rm}^* \langle\langle @ l, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \langle\langle @ m, nl_2, ol_3, e_3 \rangle\rangle :: \{\{e'_2, nl_2, ol_3, e_3\}\}\rangle\rangle.$$

Now, if $(nl_2 - m) > ol_3$, by using Lemma 6.10 repeatedly and noting that $(ol_3 \dot{-} nl_2) = 0$, it can be seen that a and b both \triangleright_{rm} -reduce to $@(m + (nl_1 \dot{-} ol_2))$. If, on the other hand, $(nl_2 - m) \leq ol_3$, we need to consider the form of $e_3[nl_2 - m]$. In the case that this is (t, p) , then, using Lemmas 6.10 and 6.7, it can be seen that both a and b \triangleright_{rm} -reduce to

$$(t, p + ((nl_2 + (nl_1 \dot{-} ol_2)) \dot{-} ol_3)).$$

If $e_3[nl_2 - m]$ is $@p$, a similar argument shows that both a and b \triangleright_{rm} -reduce to

$$@ (p + ((nl_2 + (nl_1 \dot{-} ol_2)) \dot{-} ol_3)).$$

In the other subcase, *i.e.*, when et_2 is of the form (t, m) , again using Lemma 6.10 and rule schema (m5), we see that

$$\begin{aligned} a &\triangleright_{rm}^* \langle\langle (t, m + (nl_1 \dot{-} ol_2)), nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \rangle\rangle \quad \text{and} \\ b &\triangleright_{rm}^* \langle\langle @l, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \langle\langle (t, m), nl_2, ol_3, e_3 \rangle\rangle :: \{\{e'_2, nl_2, ol_3, e_3\}\} \rangle\rangle. \end{aligned}$$

Now, if $(nl_2 - m) \geq ol_3$, by using Lemmas 6.9 and 6.10 and noting that $(ol_3 \dot{-} nl_2) = 0$, it follows that both a and b \triangleright_{rm} -reduce to $(t, m + (nl_1 \dot{-} ol_2))$. If $(nl_2 - m) < ol_3$, using Lemmas 6.9, 6.10 and 6.7, it can be seen that a and b both \triangleright_{rm} -reduce to

$$\begin{aligned} &(\llbracket t, ol_3 - (nl_2 - m), \text{ind}(e_3[nl_2 - m + 1]), e_3\{nl_2 - m + 1\} \rrbracket, \\ &\quad \text{ind}(e_3[nl_2 - m + 1]) + ((nl_2 + (nl_1 \dot{-} ol_2)) \dot{-} ol_3)). \end{aligned}$$

(b) Suppose that $et_1 = (t, l)$. We note here that $l = nl_1$. Now, let $\text{ind}(et_2) = m$. Using Lemma 6.9,

$$a \triangleright_{rm}^* \langle\langle (\llbracket t, ol_2, m, e_2 \rrbracket, m + (nl_1 \dot{-} ol_2)), nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \rangle\rangle \quad (2)$$

Our analysis now splits into two subcases, depending on whether or not $(nl_2 - m) \geq ol_3$.

Suppose that $(nl_2 - m) \geq ol_3$. Using the fact that

$$(nl_2 + (nl_1 \dot{-} ol_2) - (m + (nl_1 \dot{-} ol_2))) = (nl_2 - m) \geq ol_3$$

in conjunction with Lemma 6.9, it follows from (2) that

$$a \triangleright_{rm}^* (\llbracket t, ol_2, m, e_2 \rrbracket, m + (nl_1 \dot{-} ol_2)).$$

Now, since $(nl_2 - m) \geq ol_3$, $(ol_3 \dot{-} nl_2) = 0$. Recalling that $e_2 = et_2 :: e'_2$ and hence $\text{ind}(et_2) = \text{ind}(e_2) = m$, the following can also be seen:

$$\begin{aligned} b &= \langle\langle (t, nl_1), nl_1, ol_2, \{\{e_2, nl_2, ol_3, e_3\}\} \rangle\rangle \\ &\triangleright_{rm}^* \langle\langle (t, nl_1), nl_1, ol_2, et_2 :: e'_2 \rangle\rangle \quad (\text{Lemma 6.13}) \\ &\triangleright_{rm}^* (\llbracket t, ol_2, m, e_2 \rrbracket, m + (nl_1 \dot{-} ol_2)) \quad (\text{Lemma 6.9}). \end{aligned}$$

Thus, a and b both \triangleright_{rm} -reduce to the same expression in this case.

In the remaining subcase, $(nl_2 - m) < ol_3$. Using Lemma 6.9 in conjunction with (2), we see that

$$a \triangleright_{rm}^* (\llbracket [t, ol_2, m, e_2], ol_3 - (nl_2 - m), ind(e_3[nl_2 - m + 1]), e_3\{nl_2 - m + 1\} \rrbracket, \\ ind(e_3[nl_2 - m + 1]) + ((nl_2 + (nl_1 \dot{\div} ol_2)) \dot{\div} ol_3)).$$

Then, using rule schema (m1),

$$a \triangleright_{rm}^* (\llbracket [t, ol_2 + ((ol_3 - (nl_2 - m)) \dot{\div} m), \\ ind(e_3[nl_2 - m + 1]) + (m \dot{\div} (ol_3 - (nl_2 - m))), \\ \{\{e_2, m, ol_3 - (nl_2 - m), e_3\{nl_2 - m + 1\}\}\}, \\ ind(e_3[nl_2 - m + 1]) + ((nl_2 + (nl_1 \dot{\div} ol_2)) \dot{\div} ol_3) \rrbracket).$$

The fact that $(nl_2 - m) < ol_3$ can be used to simplify the expression on the right above. We note first that in this case

$$((ol_3 - (nl_2 - m)) \dot{\div} m) = (ol_3 \dot{\div} nl_2), \quad \text{and} \\ (m \dot{\div} (ol_3 - (nl_2 - m))) = (nl_2 \dot{\div} ol_3).$$

Thus, in fact,

$$a \triangleright_{rm}^* (\llbracket [t, ol_2 + (ol_3 \dot{\div} nl_2), ind(e_3[nl_2 - m + 1]) + (nl_2 \dot{\div} ol_3), \\ \{\{e_2, m, ol_3 - (nl_2 - m), e_3\{nl_2 - m + 1\}\}\}, \\ ind(e_3[nl_2 - m + 1]) + ((nl_2 + (nl_1 \dot{\div} ol_2)) \dot{\div} ol_3) \rrbracket). \quad (3)$$

With regard to b , we first observe, using rule schema (m5), that

$$b \triangleright_{rm} \langle\langle (t, nl_1), nl_1, ol_2 + (ol_3 \dot{\div} nl_2), \langle\langle et_2, nl_2, ol_3, e_3 \rangle\rangle :: \{\{e'_2, nl_2, ol_3, e_3\}\}\rangle\rangle.$$

Using Lemma 6.9 or 6.10 depending on the form of et_2 and recalling that $ind(et_2) = ind(e_2) = m$, we see that $\langle\langle et_2, nl_2, ol_3, e_3 \rangle\rangle \triangleright_{rm}$ -reduces to an environment term et' such that $ind(et') = ind(e_3[nl_2 - m + 1]) + (nl_2 \dot{\div} ol_3)$. But then, by Lemma 4.11, $ind(\langle\langle et_2, nl_2, ol_3, e_3 \rangle\rangle) = ind(et')$. Thus, using Lemma 6.9, we see that

$$b \triangleright_{rm}^* (\llbracket [t, ol_2 + (ol_3 \dot{\div} nl_2), ind(e_3[nl_2 - m + 1]) + (nl_2 \dot{\div} ol_3), \\ \langle\langle et_2, nl_2, ol_3, e_3 \rangle\rangle :: \{\{e'_2, nl_2, ol_3, e_3\}\}\rrbracket, \\ ind(e_3[nl_2 - m + 1]) + (nl_2 \dot{\div} ol_3) + (nl_1 \dot{\div} (ol_2 + (ol_3 \dot{\div} nl_2))) \rrbracket). \quad (4)$$

The identity of the indices of the environment terms on the right in (3) and (4) follows easily from Lemma 6.7. Inspecting these two expressions, we further see that a and b would \triangleright_{rm} -reduce to a common expression if

$$\{\{e_2, m, ol_3 - (nl_2 - m), e_3\{nl_2 - m + 1\}\}\} \quad \text{and} \quad \langle\langle et_2, nl_2, ol_3, e_3 \rangle\rangle :: \{\{e'_2, nl_2, ol_3, e_3\}\}$$

\triangleright_{rm} -reduce to a common expression. Using the rule schema (m5) and invoking Lemmas 6.11 and 6.14 after recalling that $ind(e'_2) \leq ind(et_2) = m$ and $(nl_2 - m) < ol_3$, this is easily seen to be the case.

Inductive Step for the Second Induction: $nl_1 - ind(et_1) > 0$. In this case, we observe first that, by using rule schema (m5) on b and then using rule schema (m10) on the resulting expression and on a ,

$a \triangleright_{rm}^* \langle \langle \langle et_1, nl_1 - 1, ol_2 - 1, e'_2 \rangle \rangle, nl_2 + ((nl_1 - 1) \dot{-} (ol_2 - 1)), ol_3, e_3 \rangle \rangle$, and
 $b \triangleright_{rm}^* \langle \langle et_1, nl_1 - 1, (ol_2 - 1) + (ol_3 \dot{-} nl_2), \{e'_2, nl_2, ol_3, e_3\} \rangle \rangle$.

Since $len(e'_2) < len(e_2)$, the inductive hypothesis can be invoked to conclude that the two expressions a and b are shown to \triangleright_{rm} -reduce to above themselves \triangleright_{rm} -reduce to a common expression. \square

Lemma 6.16 *Let a and b be environments of the form*

$$\begin{aligned} & \{ \{ \{ e_1, nl_1, ol_2, e_2 \} \}, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \} \quad \text{and} \\ & \{ \{ e_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{ e_2, nl_2, ol_3, e_3 \} \} \}, \end{aligned}$$

respectively. Then there is an environment r such that $a \triangleright_{rm}^* r$ and $b \triangleright_{rm}^* r$.

Proof. We assume without loss of generality that e_1 , e_2 and e_3 are simple expressions and we use an induction on $len(e_1)$.

Base Case: $len(e_1) = 0$. We note first that $e_1 = nil$. Our analysis now splits into two subcases.

(a) $nl_1 < ol_2$. Using Lemma 6.12 and noting that, in this subcase, $(nl_1 \dot{-} ol_2) = 0$, we see that

$$a \triangleright_{rm}^* \{ e_2 \{ nl_1 + 1 \}, nl_2, ol_3, e_3 \}.$$

Using rule schema (m5) repeatedly, it then follows that

$$a \triangleright_{rm}^* \langle \langle e_2[nl_1 + 1], nl_2, ol_3, e_3 \rangle \rangle :: \dots :: \langle \langle e_2[ol_2], nl_2, ol_3, e_3 \rangle \rangle :: \{ \{ nil, nl_2, ol_3, e_3 \} \}. \quad (5)$$

Now, if $nl_1 < ol_2$, then $nl_1 < (ol_2 + (ol_3 \dot{-} nl_2))$. From this and from using rule (m5) repeatedly and invoking Lemma 6.12, it follows that $b \triangleright_{rm}$ -reduces to the expression a is shown to \triangleright_{rm} -reduce to in (5).

(b) $nl_1 \geq ol_2$. By adopting arguments similar to those in subcase (a), it can be seen that a and b both \triangleright_{rm} -reduce to $e_3 \{ nl_2 + (nl_1 \dot{-} ol_2) + 1 \}$ if $ol_3 > (nl_2 + (nl_1 \dot{-} ol_2))$ and to nil if $ol_3 \leq (nl_2 + (nl_1 \dot{-} ol_2))$.

Inductive Step: $len(e_1) > 0$. In this case, e_1 is of the form $et_1 :: e'_1$. Using rule schema (m5), we see that

$$\begin{aligned} a \triangleright_{rm}^* \langle \langle \langle \langle et_1, nl_1, ol_2, e_2 \rangle \rangle \rangle, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \rangle \rangle :: \\ \{ \{ \{ e'_1, nl_1, ol_2, e_2 \} \}, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3 \} \} \end{aligned}$$

and, similarly,

$$\begin{aligned} b \triangleright_{rm}^* \langle \langle et_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{ e_2, nl_2, ol_3, e_3 \} \rangle \rangle :: \\ \{ \{ e'_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{ e_1, nl_2, ol_3, e_3 \} \} \}. \end{aligned}$$

From Lemma 6.15 and the hypothesis it now follows that a and b must \triangleright_{rm} -reduce to a common expression. \square

Theorem 6.17 *The relation \triangleright_{rm} is locally confluent.*

Proof. By virtue of Theorem 2.4, it is enough to show that, for each conflict pair $\langle r_1, r_2 \rangle$ of the rule schemata in Figures 3 and 4, there is some expression s such that $r_1 \triangleright_{rm}^* s$ and $r_2 \triangleright_{rm}^* s$. To do this, we need to consider the various nontrivial overlaps between the rule schemata in question. Examining these schemata, we see that such overlaps occur only between (m1) and each rule schema in Figure 3, (m1) and (m1) and (m2) and (m4). The last case is dealt with easily: the overlap occurs over the expression $\{\{nil, 0, 0, nil\}\}$ and the two expressions in the corresponding conflict pair are identical, both being nil . We consider the conflict pairs relative to the remaining overlaps in turn below to complete our argument. In each case, we refer to the expression that constitutes the nontrivial overlap as t and to the terms in the conflict pair as r_1 and r_2 respectively.

Overlap between (m1) and (r1). In this case t is of the form $\llbracket [c, ol_1, nl_1, e_1], ol_2, nl_2, e_2 \rrbracket$ and r_1 and r_2 are, respectively, of the form $\llbracket [c, ol', nl', \{e_1, nl_1, ol_2, e_2\}] \rrbracket$ and $\llbracket [c, ol_2, nl_2, e_2] \rrbracket$, for suitably chosen ol' and nl' . Both r_1 and r_2 obviously \triangleright_{rm} -reduce to c .

Overlap between (m1) and (r2). In this case t is of the form $\llbracket [\#i, 0, nl_1, nil], ol_2, nl_2, e_2 \rrbracket$, r_1 is of the form

$$\llbracket [\#i, ol_2 \dot{-} nl_1, nl_2 + (nl_1 \dot{-} ol_2), \{nil, nl_1, ol_2, e_2\}] \rrbracket$$

and r_2 is of the form $\llbracket [\#(i + nl_1), ol_2, nl_2, e_2] \rrbracket$. We may assume that e_2 is a simple expression, for, if it is not, it can be \triangleright_{rm} -reduced into such an expression in an identical fashion in both r_1 and r_2 and argument presented here can then be applied. Now we distinguish three cases:

$nl_1 \geq ol_2$. Using Lemmas 6.12 and 6.8 and noting that $(ol_2 \dot{-} nl_1) = 0$ and $(nl_1 \dot{-} ol_2) = (nl_1 - ol_2)$, it is easily seen that r_1 and r_2 both \triangleright_{rm} -reduce to $\#(i + nl_2 + (nl_1 - ol_2))$.

$nl_1 < ol_2$ and $i > (ol_2 - nl_1)$. A similar argument can be provided to show that r_1 and r_2 both \triangleright_{rm} -reduce to $\#(i + nl_2 - (ol_2 - nl_1))$.

$nl_1 < ol_2$ and $i \leq (ol_2 - nl_1)$. The common expression in this case depends on the form of $e[ol_1]$. If this is $@m$, then r_1 and r_2 are both \triangleright_{rm} -reduce to $\#(nl_2 - m)$ and if this is (t, m) , then r_1 and r_2 both similarly reduce to $\llbracket [t, 0, nl_2 - m, nil] \rrbracket$.

Overlap between (m1) and (r3). Here t is of the form $\llbracket [\#1, ol_1, nl_1, @l :: e_1], ol_2, nl_2, e_2 \rrbracket$, r_1 is of the form

$$\llbracket [\#1, ol_1 + (ol_2 \dot{-} nl_1), nl_2 + (nl_1 \dot{-} ol_2), \{@l :: e_1, nl_1, ol_2, e_2\}] \rrbracket,$$

and r_2 is of the form $\llbracket [\#(nl_1 - l), ol_2, nl_2, e_2] \rrbracket$. We assume without loss of generality that e_2 is a simple expression. Now we distinguish two cases:

$(nl_1 - l) > ol_2$. Using rule schema (m5), Lemmas 6.8 and 6.10 and noting that $(nl_1 \dot{-} ol_2) = (nl_1 - ol_2)$, it is easily seen that r_1 and r_2 both \triangleright_{rm} -reduce to $\#(nl_1 - l + nl_2 - ol_2)$.

$(nl_1 - l) \leq ol_2$. A similar argument can be provided to show that r_1 and r_2 \triangleright_{rm} -reduce to $\#(nl_2 - m)$ if $e_2[nl_1 - l] = @m$ and to $\llbracket [t, 0, nl_2 - m, nil] \rrbracket$ if $e_2[nl_1 - l] = (t, m)$.

Overlap between (m1) and (r4). Now t is of the form $\llbracket [\#1, ol_1, nl_1, (t, l) :: e_1], ol_2, nl_2, e_2 \rrbracket$, r_1 is of the form

$$\llbracket [\#1, ol_1 + (ol_2 \dot{-} nl_1), nl_2 + (nl_1 \dot{-} ol_2), \{(t, l) :: e_1, nl_1, ol_2, e_2\}] \rrbracket$$

and r_2 is of the form $\llbracket [t, 0, nl_1 - l, nil], ol_2, nl_2, e_2 \rrbracket$. We assume, without loss of generality, that e_2 is a simple expression. Using rule schema (m1), r_2 may be rewritten to

$$\llbracket t, ol_2 \dot{-} (nl_1 - l), nl_2 + ((nl_1 - l) \dot{-} ol_2), \{\{nil, nl_1 - l, ol_2, e_2\}\}\rrbracket.$$

From this and from using Lemmas 6.8 and 6.12, it is easily seen that in the case that $(nl_1 - l) \geq ol_2$, r_1 and r_2 both \triangleright_{rm} -reduce to $\llbracket t, 0, nl_2 + (nl_1 - l) - ol_2, nil \rrbracket$. In the case that $(nl_1 - l) < ol_2$, using Lemma 6.12 we see first that r_2 \triangleright_{rm} -reduces to

$$\llbracket t, ol_2 - (nl_1 - l), nl_2, e_2\{nl_1 - l + 1\} \rrbracket.$$

Further, using rule schema (m5) and Lemma 6.9 and letting $ind(e_2\{nl_1 - l + 1\}) = m$,

$$\begin{aligned} & \{\{t, l\} :: e_1, nl_1, ol_2, e_2\} \triangleright_{rm}^* \\ & (\llbracket t, ol_2 - (nl_1 - l), m, e_2\{nl_1 - l + 1\} \rrbracket, m + (nl_1 \dot{-} ol_2)) :: \{\{e_1, nl_1, ol_2, e_2\}\}. \end{aligned}$$

Thus, additionally using Lemma 6.8,

$$r_1 \triangleright_{rm}^* \llbracket \llbracket t, ol_2 - (nl_1 - l), m, e_2\{nl_1 - l + 1\} \rrbracket, 0, nl_2 - m, nil \rrbracket.$$

But now, using rule schema (m1) and invoking Lemma 6.13, it is easily seen that the expression on the right \triangleright_{rm} -reduces to $\llbracket t, ol_2 - (nl_1 - l), nl_2, e_2\{nl_1 - l + 1\} \rrbracket$. Thus, once again, r_1 and r_2 \triangleright_{rm} -reduce to a common expression.

Overlap between (m1) and (r5). Here t is of the form $\llbracket \llbracket \#k, ol_1, nl_1, et :: e_1 \rrbracket, ol_2, nl_2, e_2 \rrbracket$ where $k > 1$, r_1 is of the form

$$\llbracket \#k, ol_1 + (ol_2 \dot{-} nl_1), nl_2 + (nl_1 \dot{-} ol_2), \{\{et :: e_1, nl_1, ol_2, e_2\}\} \rrbracket$$

and r_2 is of the form $\llbracket \llbracket \#(k - 1), ol_1 - 1, nl_1, e_1 \rrbracket, ol_2, nl_2, e_2 \rrbracket$. It is easily seen in this case that r_1 and r_2 both \triangleright_{rm} -reduce to an expression of the form

$$\llbracket \#(k - 1), ol_1 + (ol_2 \dot{-} nl_1) - 1, nl_2 + (nl_1 \dot{-} ol_2), \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket.$$

Overlap between (m1) and (r6). Here t is of the form $\llbracket \llbracket (t_1 \ t_2), ol_1, nl_1, e_1 \rrbracket, ol_2, nl_2, e_2 \rrbracket$, r_1 is of the form $\llbracket (t_1 \ t_2), ol', nl', \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket$ where $ol' = (ol_1 + (ol_2 \dot{-} nl_1))$ and $nl' = (nl_2 + (nl_1 \dot{-} ol_2))$, and r_2 is of the form $\llbracket \llbracket (t_1, ol_1, nl_1, e_1) \rrbracket \llbracket (t_2, ol_1, nl_1, e_1) \rrbracket, ol_2, nl_2, e_2 \rrbracket$. It is easily seen that r_1 and r_2 both \triangleright_{rm} -reduce to an expression of the form

$$(\llbracket t_1, ol', nl', \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket \llbracket t_2, ol', nl', \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket).$$

Overlap between (m1) and (r7). In this case t is of the form $\llbracket \llbracket (\lambda t'), ol_1, nl_1, e_1 \rrbracket, ol_2, nl_2, e_2 \rrbracket$, r_1 is of the form $\llbracket (\lambda t'), ol', nl', \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket$ where $ol' = (ol_1 + (ol_2 \dot{-} nl_1))$ and $nl' = (nl_2 + (nl_1 \dot{-} ol_2))$, and r_2 is of the form

$$\llbracket (\lambda \llbracket t', ol_1 + 1, nl_1 + 1, @nl_1 :: e_1 \rrbracket), ol_2, nl_2, e_2 \rrbracket.$$

Now, using rule schema (r7), we see that

$$r_1 \triangleright_{rm}^* (\lambda \llbracket t', ol' + 1, nl' + 1, @nl' :: \{\{e_1, nl_1, ol_2, e_2\}\} \rrbracket).$$

Similarly, using rule schemata (r7), (m1) and (m5) and invoking Lemma 6.10, we observe that

$$r_2 \triangleright_{rm}^* (\lambda [t', ol' + 1, nl' + 1, @nl' :: \{\{e_1, nl_1 + 1, ol_2 + 1, @nl_2 :: e_2\}\}]).$$

Noting then that $ind(e_1) \leq nl_1$ and using Lemma 6.14, we conclude that

$$\{\{e_1, nl_1 + 1, ol_2 + 1, @nl_2 :: e_2\}\} \quad \text{and} \quad \{\{e_1, nl_1, ol_2, e_2\}\}$$

\triangleright_{rm} -reduce to a common expression. But then r_1 and r_2 also \triangleright_{rm} -reduce to a common expression.

Overlap between (m1) and (m1). Here t is of the form

$$\llbracket \llbracket [t_1, ol_1, nl_1, e_1], ol_2, nl_2, e_2 \rrbracket, ol_3, nl_3, e_3 \rrbracket$$

and r_1 and r_2 are of the form

$$\begin{aligned} & \llbracket [t_1, ol_1, nl_1, e_1], ol_2 + (ol_3 \dot{-} nl_2), nl_3 + (nl_2 \dot{-} ol_3), \{\{e_2, nl_2, ol_3, e_3\}\} \rrbracket \quad \text{and} \\ & \llbracket [t_1, ol_1 + (ol_2 \dot{-} nl_1), nl_2 + (nl_1 \dot{-} ol_2), \{\{e_1, nl_1, ol_2, e_2\}\}], ol_3, nl_3, e_3 \rrbracket. \end{aligned}$$

Using rule schema (m1), we see that

$$r_1 \triangleright_{rm}^* \llbracket [t_1, ol', nl', \{\{e_1, nl_1, ol_2 + (ol_3 \dot{-} nl_2), \{\{e_2, nl_2, ol_3, e_3\}\}\} \rrbracket \rrbracket \quad (6)$$

where $ol' = ol_1 + ((ol_2 + (ol_3 \dot{-} nl_2)) \dot{-} nl_1)$ and $nl' = nl_3 + (nl_2 \dot{-} ol_3) + (nl_1 \dot{-} (ol_2 + (ol_3 \dot{-} nl_2)))$. Similarly,

$$r_2 \triangleright_{rm}^* \llbracket [t_1, ol'', nl'', \{\{\{e_1, nl_1, ol_2, e_2\}\}, nl_2 + (nl_1 \dot{-} ol_2), ol_3, e_3\}\} \rrbracket \quad (7)$$

where $ol'' = ol_1 + (ol_2 \dot{-} nl_1) + (ol_3 \dot{-} (nl_2 + (nl_1 \dot{-} ol_2)))$ and $nl'' = nl_3 + ((nl_2 + (nl_1 \dot{-} ol_2)) \dot{-} ol_3)$.

Using Lemma 6.16 in conjunction with (6) and (7), we see that r_1 and r_2 would \triangleright_{rm} -reduce to a common expression if $ol' = ol''$ and $nl' = nl''$. But this is easily seen to be the case by using Lemma 6.7.

All the necessary cases have been considered and the proof of the theorem is, thus, complete. \square

The following theorem is an immediate consequence of Proposition 2.2, Theorem 6.6, and Theorem 6.17.

Theorem 6.18 *The reduction relation \triangleright_{rm} is confluent.*

By virtue of Theorem 6.6, Proposition 2.1 and Theorem 6.18, we see that every suspension expression has a unique \triangleright_{rm} -normal form. It will be convenient to have a special notation for such forms. Such a notation is introduced by the following definition.

Definition 6.19 The \triangleright_{rm} -normal form of an expression t is denoted by $|t|$.

The discussion at the outset of this section can now be given formal content: The purpose of the reading and merging rules can be understood as that of incrementally producing $|t|$ from any given expression t .

7 Correspondence to beta reduction on de Bruijn terms

The β_s -contraction rule schema is intended to be a counterpart in the context of suspension terms of the β -contraction rule schema for de Bruijn terms. The correspondence between these two schemata can be made more precise at this stage. The reading and merging rules, as we have seen, partition the collection of suspension terms into equivalence classes based on the notion of “having the same \triangleright_{rm} -normal form.” The intention, then, is that the β_s -contraction rule schema have the same effect relative to the *equivalence* classes of suspension terms as does the β -contraction rule schema relative to de Bruijn terms. We show, in this section, that this is indeed the case. In particular, we demonstrate that a use of one of these schemata in one context can be mimicked by some sequence of applications of the other schemata in the other context.

As we have observed before, a suspension term is intended to encapsulate a de Bruijn term with a ‘pending’ substitution. It is useful to state this correspondence precisely. The following lemma does this using the meta-notation for substitution described in Section 3.

Lemma 7.1 *Let $t = \llbracket t', ol, nl, e \rrbracket$ be a term and let $e' = |e|$. Then $|t| = S(|t'|; s_1, s_2, s_3, \dots)$ where*

$$s_i = \begin{cases} \#(i - ol + nl) & \text{if } i > ol \\ \#(nl - m) & \text{if } i \leq ol \text{ and } e'[i] = @m \\ \llbracket t_i, 0, nl - m, nil \rrbracket & \text{if } i \leq ol \text{ and } e'[i] = (t_i, m). \end{cases}$$

Proof. By induction on t with respect to the well founded ordering relation \succ . The argument is based on a consideration of the structure of the term t' .

If t' is a constant: In this case $|t|$ and $S(|t'|; s_1, s_2, s_3, \dots)$ are both identical to t' .

If t' is a variable reference: Noting the confluence of \triangleright_{rm} , the desired conclusion in this case follows easily from Lemma 6.8.

If t' is an application: Let $t' = (r_1 r_2)$. Now $t \triangleright_{rm} (\llbracket r_1, ol, nl, e \rrbracket \llbracket r_2, ol, nl, e \rrbracket)$ by virtue of rule schema (r6) and, therefore, by the confluence of \triangleright_{rm} ,

$$|t| = (|\llbracket r_1, ol, nl, e \rrbracket \llbracket r_2, ol, nl, e \rrbracket|). \quad (1)$$

Additionally, using Lemma 6.5, $t \succ (\llbracket r_1, ol, nl, e \rrbracket \llbracket r_2, ol, nl, e \rrbracket)$. Now, for $i = 1$ and $i = 2$,

$$(\llbracket r_1, ol, nl, e \rrbracket \llbracket r_2, ol, nl, e \rrbracket) \succ \llbracket r_i, ol, nl, e \rrbracket$$

and, by transitivity, $t \succ \llbracket r_i, ol, nl, e \rrbracket$. Invoking the hypothesis of the induction,

$$|\llbracket r_i, ol, nl, e \rrbracket| = S(|r_i|; s_1, s_2, s_3, \dots).$$

From this fact used in conjunction with (1) and Definition 3.2, it follows that

$$|t| = S((|r_1| |r_2|); s_1, s_2, s_3, \dots).$$

Noting finally that $|t'| = (|r_1| |r_2|)$, the lemma is seen to hold in this case.

If t' is an abstraction: Let $t' = (\lambda r)$. Then $|t| = (\lambda |\llbracket r, ol + 1, nl + 1, @nl :: e \rrbracket|)$ by virtue of rule schema (r7) and the confluence of \triangleright_{rm} . By an argument similar to that employed in the case that t' is an application, we also see that $t \succ \llbracket r, ol + 1, nl + 1, @nl :: e \rrbracket$. Using the inductive hypothesis, $|\llbracket r, ol + 1, nl + 1, @nl :: e \rrbracket| = S(|r|; s'_1, s'_2, s'_3, \dots)$ where

$$s'_i = \begin{cases} \#1 & \text{if } i = 1 \\ \#(i - ol + nl) & \text{if } i > ol + 1 \\ \#(nl + 1 - m) & \text{if } 1 < i \leq (ol + 1) \text{ and } e'[i - 1] = @m \\ |[s, 0, nl + 1 - m, nil]| & \text{if } 1 < i \leq (ol + 1) \text{ and } e'[i - 1] = (s, m) \end{cases} \quad (2)$$

Noting now that $|t'| = (\lambda|r|)$ and using Definition 3.2, we see that

$$S(|t'|; s_1, s_2, s_3, \dots) = (\lambda S(|r|; \#1, S(s_1; \#2, \#3, \#4, \dots), S(s_2; \#2, \#3, \#4, \dots), \dots)). \quad (3)$$

From inspecting (2) and (3), it follows that the lemma would hold in this case if, for $i \geq 1$, $s'_{i+1} = S(s_i; \#2, \#3, \#4, \dots)$. We show that this must be true by considering several subcases.

- (a) $i > ol$. In this case both terms are $\#(i + 1 - ol + nl)$ and hence are identical.
- (b) $1 < i \leq ol$ and $e'[i]$ is of the form $@m$. Now both terms are identical to $\#(nl + 1 - m)$.
- (c) $1 < i \leq ol$ and $e'[i]$ is of the form (s, m) . Here we need to show that

$$|[s, 0, nl + 1 - m, nil]| = S(|[s, 0, nl - m, nil]|; \#2, \#3, \#4, \dots). \quad (4)$$

By virtue of rule schemata (m1) and (m2), $[[[s, 0, nl - m, nil], 0, 1, nil] \triangleright_{rm}^* [s, 0, nl + 1 - m, nil]]$ and, thus,

$$|[s, 0, nl + 1 - m, nil]| = |[[[s, 0, nl - m, nil], 0, 1, nil]]|. \quad (5)$$

Referring to Definition 5.1, we claim that $\eta(t) > \eta([[[s, 0, nl - m, nil], 0, 1, nil]])$. This is seen by noting the following: $\eta([[[s, 0, nl - m, nil], 0, 1, nil]]) = \mu(s) + 1$, $\eta(t) \geq \mu(s) + \mu((\lambda r))$, and $\mu((\lambda r)) \geq 2$. It thus follows that $t \succ [[[[s, 0, nl - m, nil], 0, 1, nil]]]$. The inductive hypothesis can therefore be applied to the term on the right of (5). Doing so easily yields (4).

If t' is a suspension: Using Lemma 6.5 and noting that $t' \neq |t'|$, $t \succ [|t'|, ol, nl, \epsilon]$. Invoking the inductive hypothesis with respect to the latter term and noting that $||t'| = |t'|$, the lemma follows in this case. □

We now show the desired correspondence in one of the two directions. This observation may be viewed as a relative completeness result for the β_s -contraction rule schema.

Lemma 7.2 *Let t be a de Bruijn term and let $t \triangleright_{\beta_s}$. Then there is a suspension term r such that $t \triangleright_{\beta_s} r$ and $|r| = s$.*

Proof. By an induction on the structure of t .

Base Case: t is the β -redex rewritten by a β -contraction rule. Let $t = ((\lambda t_1) t_2)$. By definition,

$$s = S(t_1; t_2, \#1, \#2, \dots). \quad (6)$$

Now let $r = [|t_1, 1, 0, (t_2, 0) :: nil]|$. Obviously $t \triangleright_{\beta_s} r$ and, using Lemma 7.1,

$$|r| = S(|t_1|; |[t_2, 0, 0, nil]|, \#1, \#2, \dots). \quad (7)$$

Noting that t_1 is a de Bruijn term, it follows that $|t_1| = t_1$. Using Lemma 7.1 and noting that t_2 is a de Bruijn term, we similarly see that $|\llbracket t_2, 0, 0, nil \rrbracket| = t_2$. Thus, the terms on the righthand sides of (6) and (7) are identical, *i.e.*, $|r| = s$.

Inductive Step: t is an abstraction or an application. The argument in both cases is similar so we consider only the first case. Let $t = (\lambda t_1)$. Then $s = (\lambda s_1)$ where s_1 is such that $t_1 \triangleright_{\beta} s_1$. By hypothesis, there is a suspension term r_1 such that $t_1 \triangleright_{\beta_s} r_1$ and $|r_1| = s_1$. Letting $r = (\lambda r_1)$, we see that the requirements of the lemma are satisfied: $|r| = (\lambda |r_1|) = (\lambda s_1) = s$ and obviously $t \triangleright_{\beta_s} r$. □

It is useful to extend the above lemma to suspension expressions. For this purpose, it is necessary to consider the use of the β -contraction rule schema on such expressions.

Definition 7.3 The relation on suspension expressions generated by the β -contraction rule schema is denoted by $\triangleright_{\beta'}$.

We observe that when restricted to suspension terms in \triangleright_{rm} -normal form, *i.e.*, to de Bruijn terms, $\triangleright_{\beta'}$ is identical to \triangleright_{β} . The following lemmas, whose proofs are obvious, ensure that $\triangleright_{\beta'}$ preserves the length of an environment and the index of an environment and an environment term. Thus, $\triangleright_{\beta'}$ is well defined, *i.e.*, it relates a given well formed expression only to another well formed one.

Lemma 7.4 *Let et_1 be an environment term and let et_2 be such that $et_1 \triangleright_{\beta'}^* et_2$. Then the following holds: if et_1 is $@m$, then et_2 is $@m$; if et_1 is of the form (t_1, m) , then et_2 is of the form (t_2, m) . Further, if et_1 is in \triangleright_{rm} -normal form, then, in the latter case, $t_1 \triangleright_{\beta}^* t_2$.*

Lemma 7.5 *Let e_1 be an environment and let e_2 be such that $e_1 \triangleright_{\beta'}^* e_2$. Then $\text{len}(e_1) = \text{len}(e_2)$. Further, if $\text{len}(e_1) > 0$, then the following holds for $1 \leq i \leq \text{len}(e_1)$: if $e_1[i]$ is $@m$, then $e_2[i]$ is $@m$; if $e_1[i]$ is of the form (t_1, m) , then $e_2[i]$ is of the form (t_2, m) . Finally, if e_1 is in \triangleright_{rm} -normal form, then, in the latter case, $t_1 \triangleright_{\beta}^* t_2$.*

The following lemma that strengthens Lemma 7.2 follows from it by an easy induction on the structure of suspension expressions.

Lemma 7.6 *Let x and y be suspension expressions such that $x \triangleright_{\beta'} y$. Then there is a suspension expression z such that $x \triangleright_{\beta_s} z$ and $|z| = |y|$.*

The coupling that Lemma 7.2 yields in the direction considered is fairly tight: it shows that each application of the β -contraction rule schema on de Bruijn terms can be mimicked by a *single* use of the β_s -contraction rule schema and some reading and merging steps. The correspondence in the converse direction is not as strict: mimicking an application of the β_s -contraction rule schema may require *several* or *no* uses of the β -contraction rule schema on the underlying de Bruijn term. This is a reflection of the fact that using environments may foster a sharing of β -redexes or,

alternatively, may result in temporarily maintaining β -redexes that would not appear in the term if the substitution were carried out completely.

The next three lemmas provide some insight into the nature of the \triangleright_{β_s} -reduction relation and are also useful in establishing the correspondence in the remaining direction.

Lemma 7.7 *Let t_1 be a term in \triangleright_{rm} -normal form and let t_2 be such that $t_1 \triangleright_{\beta'}^* t_2$. Further, let e_1 be an environment in \triangleright_{rm} -normal form and let e_2 be such that $e_1 \triangleright_{\beta'}^* e_2$. Then*

$$|\llbracket t_1, ol, nl, e_1 \rrbracket| \triangleright_{\beta'}^* |\llbracket t_2, ol, nl, e_2 \rrbracket|.$$

Proof. We note, using Lemma 7.1 and Corollary 3.6, that if s_1 and s_2 are de Bruijn terms such that $s_1 \triangleright_{\beta'}^* s_2$, then

$$|\llbracket s_1, 0, n, nil \rrbracket| \triangleright_{\beta'}^* |\llbracket s_2, 0, n, nil \rrbracket|.$$

Using Lemmas 7.1 and 7.5 in conjunction with the assumptions of the lemma, it follows easily that

$$|\llbracket t_1, ol, nl, e_1 \rrbracket| = S(u_0; u_1, u_2, u_3, \dots) \quad \text{and} \quad |\llbracket t_2, ol, nl, e_2 \rrbracket| = S(v_0; v_1, v_2, v_3, \dots),$$

where, for $i \geq 0$, $u_i \triangleright_{\beta'}^* v_i$. But then, by Corollary 3.6, $|\llbracket t_1, ol, nl, e_1 \rrbracket| \triangleright_{\beta'}^* |\llbracket t_2, ol, nl, e_2 \rrbracket|$. □

Lemma 7.8 *Let et_1 be an environment term in \triangleright_{rm} -normal form and let et_2 be an expression such that $et_1 \triangleright_{\beta'}^* et_2$. Further, let e_1 be an environment in \triangleright_{rm} -normal form and let e_2 be such that $e_1 \triangleright_{\beta'}^* e_2$. Then $|\llbracket \langle et_1, nl, ol, e_1 \rangle \rrbracket| \triangleright_{\beta'}^* |\llbracket \langle et_2, nl, ol, e_2 \rangle \rrbracket|$.*

Proof. An easy consequence of Lemmas 7.4, 7.5, 6.9, 6.10 and 7.7. □

Lemma 7.9 *Let e_1 and e_2 be environments in \triangleright_{rm} -normal form and let e'_1 and e'_2 be such that $e_1 \triangleright_{\beta'}^* e'_1$ and $e_2 \triangleright_{\beta'}^* e'_2$. Then $|\llbracket \langle e_1, nl, ol, e_2 \rangle \rrbracket| \triangleright_{\beta'}^* |\llbracket \langle e'_1, nl, ol, e'_2 \rangle \rrbracket|$.*

Proof. By induction on $len(e_1)$. If $len(e_1) = 0$, then e_1 and e'_1 are both *nil*. Then, by Lemma 6.12, either $|\llbracket \langle e_1, nl, ol, e_2 \rangle \rrbracket|$ and $|\llbracket \langle e'_1, nl, ol, e'_2 \rangle \rrbracket|$ are both *nil*, or, for some k , $|\llbracket \langle e_1, nl, ol, e_2 \rangle \rrbracket| = e_2\{k\}$ and $|\llbracket \langle e'_1, nl, ol, e'_2 \rangle \rrbracket| = e'_2\{k\}$. The desired conclusion follows easily in either case. If $len(e_1) > 0$, let $e_1 = et_1 :: t_1$, noting that et_1 and t_1 must be in \triangleright_{rm} -normal form. Then e'_1 must be of the form $et'_1 :: t'_1$ where $et_1 \triangleright_{\beta'}^* et'_1$ and $t_1 \triangleright_{\beta'}^* t'_1$. Using rule schema (m5),

$$\begin{aligned} |\llbracket \langle e_1, nl, ol, e_2 \rangle \rrbracket| &= |\llbracket \langle et_1, nl, ol, e_2 \rangle \rrbracket| :: |\llbracket \langle t_1, nl, ol, e_2 \rangle \rrbracket|, \quad \text{and} \\ |\llbracket \langle e'_1, nl, ol, e'_2 \rangle \rrbracket| &= |\llbracket \langle et'_1, nl, ol, e'_2 \rangle \rrbracket| :: |\llbracket \langle t'_1, nl, ol, e'_2 \rangle \rrbracket|. \end{aligned}$$

The lemma now follows from Lemma 7.8 and the inductive hypothesis. □

We now show the converse correspondence that may be viewed as a relative soundness result for the β_s -contraction rule schema. Recalling that $\triangleright_{\beta'}$ is identical to \triangleright_{β} when restricted to de Bruijn terms, we see that this observation is contained in the following lemma.

Lemma 7.10 *Let t and s be suspension expressions such that $t \triangleright_{\beta_s} s$. Then $|t| \triangleright_{\beta'}^* |s|$.*

Proof. The proof is by induction on t with respect to \succ . To begin with, we note that t cannot be a constant, a variable reference, nil or of the form $@m$. We consider now the remaining cases for the structure of t .

If t is an application: There are two possibilities: t is the redex rewritten by a β_s -contraction rule or some proper subterm of t is rewritten. We analyze each possibility separately.

In the first subcase, t has the form $((\lambda t_1) t_2)$. We note first that $|t| = ((\lambda |t_1|) |t_2|)$. Further,

$$s = [|t_1, 1, 0, (t_2, 0) :: nil|].$$

Using Lemma 7.1, it can be seen that $|s| = S(|t_1|; |t_2|, \#1, \#2, \dots)$, *i.e.*, that $|t| \triangleright_{\beta'} |s|$.

In the second subcase, t is of the form $(t_1 t_2)$. We assume, without loss of generality, that the redex rewritten is a subterm of t_1 . Then $s = (s_1 t_2)$, where $t_1 \triangleright_{\beta_s} s_1$. Since t_1 is a proper subterm of t , $t \succ t_1$. Thus, by hypothesis, $|t_1| \triangleright_{\beta'}^* |s_1|$. The lemma now follows from noting that $|t| = (|t_1| |t_2|)$ and $|s| = (|s_1| |t_2|)$.

If t is an abstraction or has the form (t', m) or $et :: e$: A straightforward inductive argument, similar to that provided for the second subcase of the application case, suffices in each of these cases.

If t is a suspension: Let $t = [|r, ol, nl, e|]$. Then $s = [|r', ol, nl, e'|]$ where $r \triangleright_{\beta_s} r'$ and $e = e'$ or $r = r'$ and $e \triangleright_{\beta_s} e'$. In either case, using the fact that r and e are proper subexpressions of t and hence $t \succ r$ and $t \succ e$, $|r| \triangleright_{\beta'}^* |r'|$ and $|e| \triangleright_{\beta'}^* |e'|$. Now, by confluence of \triangleright_{rm}^* ,

$$[|r, ol, nl, e|] = [| [|r|, ol, nl, |e| |]] \quad \text{and} \quad [|r', ol, nl, e'|] = [| [|r'|, ol, nl, |e'| |]].$$

Using Lemma 7.7, it follows from this that $[|r, ol, nl, e|] \triangleright_{\beta'}^* [|r', ol, nl, e'|]$. Recalling that $\triangleright_{\beta'}$ and \triangleright_{β} are identical on de Bruijn terms, the lemma is seen to be true.

If t has the form $\langle\langle et, nl, ol, e \rangle\rangle$: By an argument similar to that used for a suspension, s must be of the form $\langle\langle et', ol, nl, e' \rangle\rangle$ where $|et| \triangleright_{\beta'}^* |et'|$ and $|e| \triangleright_{\beta'}^* |e'|$. The lemma now follows from noting that

$$|\langle\langle et, nl, ol, e \rangle\rangle| = |\langle\langle |et|, nl, ol, |e| \rangle\rangle| \quad \text{and} \quad |\langle\langle et', nl, ol, e' \rangle\rangle| = |\langle\langle |et'|, nl, ol, |e'| \rangle\rangle|,$$

and using Lemma 7.8.

If t is of the form $\{\{e_1, nl, ol, e_2\}\}$: Once again, s must be of the form $\{\{e'_1, nl, ol, e'_2\}\}$ where, $|e_1| \triangleright_{\beta'}^* |e'_1|$ and $|e_2| \triangleright_{\beta'}^* |e'_2|$. We note further that

$$|\{\{e_1, nl, ol, e_2\}\}| = |\{\{|e_1|, nl, ol, |e_2|\}\}| \quad \text{and} \quad |\{\{e'_1, nl, ol, e'_2\}\}| = |\{\{|e'_1|, nl, ol, |e'_2|\}\}|.$$

The lemma now follows from Lemma 7.9.

All possibilities for the structure of t having been considered, the proof of the lemma is complete. □

8 Some reduction properties of the overall system

We now use the results of the previous two sections to observe some simple properties of reduction within our system of rewrite rules. The first observation we make is that the rule schemata in Figures 2–4 correctly implement β -reduction. A generalization of this observation is contained in the following theorem.

Theorem 8.1

- (a) If x and y are suspension expressions such that $x \triangleright_{rm\beta_s}^* y$, then $|x| \triangleright_{\beta'}^* |y|$.
- (b) If x and y are suspension expressions in \triangleright_{rm} -normal form such that $x \triangleright_{\beta'}^* y$ then $x \triangleright_{rm\beta_s}^* y$.

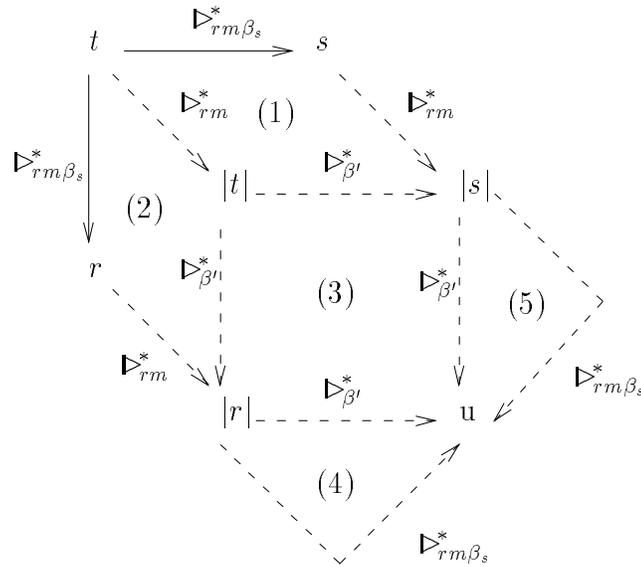
Proof. (a) By an induction on the length of the reduction sequence by which $x \triangleright_{rm\beta_s}^* y$. If the first rule used is an instance of the β_s -contraction rule schema, we use Lemma 7.10. Otherwise we use Theorem 6.18 to note that the \triangleright_{rm} -normal form is preserved.

(b) An induction on the length of the reduction sequence by which $x \triangleright_{\beta'}^* y$. It is only necessary to show that if $x \triangleright_{\beta'}^* y$, then $x \triangleright_{rm\beta_s}^* y$. This follows from Lemma 7.6 by noting that y must be in \triangleright_{rm} -normal form and using the fact that \triangleright_{rm} is confluent and noetherian. □

We also note that the overall system of rewrite rules is confluent.

Theorem 8.2 *The reduction relation $\triangleright_{rm\beta_s}$ is confluent.*

Proof. This is evident from the diagram below.



In this diagram, the (existential) dashed arrows in the faces (1) and (2) are justified by Theorem 8.1, the remaining dashed arrows in face (3) are justified by a straightforward extension of Proposition 3.7 to $\triangleright_{\beta'}^*$ and the last two dashed arrows in faces (4) and (5) are justified by Theorem 8.1. □

Another observation we make concerns the ability to eliminate the merging rules in certain contexts. As we have noted, these rules are useful in combining substitution walks over terms and thus have certain efficiency advantages. However, as we show now, they are not necessary if the sole purpose is that of implementing β -reduction.

Lemma 8.3 *Let t be a de Bruijn term and let $t \triangleright_{\beta} s$. Then $t \triangleright_{r\beta_s}^* s$.*

Proof. By Lemma 7.2, $t \triangleright_{\beta_s} r$ where $|r| = s$. We observe now that t , being a de Bruijn term, is a simple expression. From this it follows that r is also a simple expression. It is also easily seen that (a) a reading rule must be applicable to any simple expression that is not in \triangleright_{rm} -normal form, and (b) applying such a rule produces another simple expression. Thus $r \triangleright_r^* |r|$, i.e., $r \triangleright_r^* s$. This implies that $t \triangleright_{r\beta_s}^* s$. □

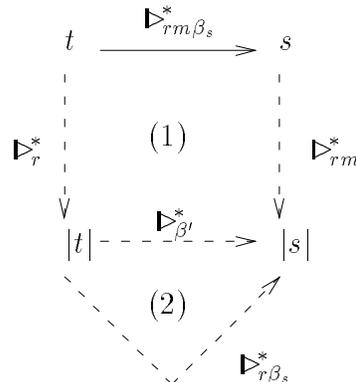
Lemma 8.4 *Let t and s be de Bruijn terms such that $t \triangleright_{\beta} s$. Then $t \triangleright_{r\beta_s}^* s$.*

Proof. By induction on the length of the \triangleright_{β} -reduction sequence, using Lemma 8.3. □

It is not possible, in a general sense, to eliminate uses of merging rules from a $\triangleright_{rm\beta_s}$ -reduction sequence. However, when starting from a simple expression, merging rules are redundant if the objective is to produce an expression in the same ‘equivalence class’ relative to the \triangleright_{rm} relation as the final expression that was originally produced.

Lemma 8.5 *Let t be a simple expression and let s be such that $t \triangleright_{rm\beta_s}^* s$. Then there is an expression u such that $t \triangleright_{r\beta_s}^* u$ and $s \triangleright_{rm}^* u$.*

Proof. Letting u be the expression $|s|$, the lemma is evident from the diagram below.



The dashed arrows in the face labelled (1) in this figure are justified by Theorem 8.1; the label \triangleright_r^* on the arrow from t to $|t|$ is warranted by the observation (made in the proof of Lemma 8.3) that a simple expression can be reduced to its \triangleright_{rm} -normal form by using only reading rules. The remaining dashed arrow in face (2) is justified by Lemma 8.4. □

Several other observations can be made of the overall rewrite system and this is, in fact, done in a companion paper [Nad94] where we consider refinements to and applications of the rewrite system presented here. The present observations are of interest because they shed light on the interaction between rules in our system. Moreover, they show the usefulness of the results of the previous two sections and the consequent ‘projection’ onto de Bruijn terms in establishing properties of our system. As already mentioned, this idea is similar in spirit to the one referred to as the *interpretation method* in [Har89] and used in [Har89] and [Yok89] in proving confluence properties of a combinator calculus. We utilize this idea again in [Nad94].

9 Conclusion

We have described in this paper a notation for the terms in a lambda calculus and a system for rewriting expressions in this notation. Our notation is based on the de Bruijn representation of lambda terms but embellishes this so as to allow for the representation of a term with a pending substitution. We have shown that the rewrite rules in our system can simulate the operation of β -reduction on terms in the usual representation and can, in a sense, be simulated by this operation. We have used this observation in establishing the confluence of our overall system. The notation developed here has several useful features. It is closely related to the usual representation of lambda terms and can in fact replace the latter notation even in contexts where intensions of terms have to be manipulated. The use of de Bruijn’s scheme for representing variables obviates α -conversion in comparing terms. Our rewrite system provides a fine-grained control over the substitution process involved in β -contraction, and thus can be used as the basis for a wide variety of reduction procedures. Furthermore, the ability our notation provides to suspend substitutions leads to efficiency advantages in the implementation of β -reduction: substitution and reduction walks over the structures of terms can be combined and substitutions can be delayed in some cases till such a point that it becomes unnecessary to perform them. Finally, our notation permits components of a β -contraction step to be intermingled with other operations such as those involved in unifying lambda terms. This ability is of practical relevance and is, in fact, used to advantage in an implementation of the language λ Prolog.

While the specific notation presented here is new, the ideas embedded in it have received previous and parallel developments and it is relevant to comment on this aspect. A central idea in our notation is the use of environments in representing suspended substitutions. This idea is an old one within the implementation of β -reduction to the extent that it is difficult to pinpoint a source for it. The category of terms that we have referred to as suspensions in this paper are what are usually called *closures*. However, most of these proposals have differed from that presented

in this paper in two important respects. First, the idea of closures has been used largely as an implementation device and an attempt has not been made to reflect it into the notation or to describe a calculus that takes the resulting notation seriously. Second, in most cases the focus has been on generating *weak head normal forms*, *i.e.*, the percolation of substitutions or the rewriting of β -redexes under abstractions is not considered. The latter assumption has the effect of greatly simplifying the kind of notation required, as the reader may well verify. Moreover, as discussed already, this is not an assumption that is valid in all contexts.

In our knowledge, the first serious consideration of a notation and a calculus that incorporate a fine-grained control over substitutions appears in the work of P-L. Curien [Cur86a, Cur86b]. In this work, a categorical combinatory logic called **CCL** is described. The language underlying this logic is not the lambda calculus, but bears a close relationship to it: there is a translation from the (pure) lambda calculus augmented with the pairing function to **CCL** and vice versa that preserves the intended equality relation in the two calculi. Unfortunately, the rewrite rules that constitute **CCL** are not confluent [Har89]; this result might be anticipated from the fact that the lambda calculus with the pairing function is not confluent [Klo80]. However, a subset of **CCL** terms can be exhibited on which the rewrite rules are confluent [Har89, Yok89]. Moreover, a subclass of this class of terms is isomorphic to the class of lambda calculus terms and this isomorphism can be extended to one between a subset of **CCL** rules and β -reduction [Har89]. An interesting characteristic of this subsystem is that it permits ' β -contraction' to be factored into the generation of a substitution and the subsequent percolation of this substitution in much the spirit of the system described in this paper.

While the **CCL** system has several desirable features, its relationship to the lambda calculus is a somewhat complex one. More recently, the general ideas embedded in **CCL** have been used in conjunction with notations that are more directly based on the lambda calculus in [ACCL90] and [Fie90]. The resulting systems are very similar to the one described here and our work, in fact, represents a concurrent and independent development of these general ideas.⁵ At a level of detail, the notations in [ACCL90] and [Fie90] are practically indistinguishable. However, they differ from our notation in two respects. The first of these is in the manner in which variables are represented. In our notation, these are represented directly by de Bruijn numbers. In contrast, in the other notations, variables are represented essentially as environment transforming operators that strip off parts of environments. The latter representation has the virtue of parsimony: a smaller vocabulary suffices and the rules that serve to combine environments can also be used to determine the bindings for variables. However, there are also advantages to our representation. As one example, the comparison of terms containing variables becomes somewhat easier. At a different level, there is a differentiation of rules in our system based on purpose, and this makes it easier to identify simpler, but yet complete, subsystems. Thus, as observed in Lemma 8.4, the rules for merging environments can be omitted from our system without losing the ability to simulate β -reduction. A similar observation cannot be made about the other systems being discussed.⁶

⁵The ideas described here are an outgrowth of those contained in [NW90]. The present exposition of these ideas has, however, been influenced by [ACCL90].

⁶We note in this context that the remark in [ACCL90] to the effect that the rule for merging environments (labelled

The second respect in which our notation differs from the ones in [ACCL90] and [Fie90] is the manner in which it encodes the adjustment that must be made to indices of terms in an environment. In our notation, this is not maintained explicitly but is obtained from the difference between the embedding level of the term that has to be substituted into and an embedding level recorded with the term in the environment. Thus, consider a suspension term of the form $\llbracket t_1, 1, nl, (t_2, nl') :: nil \rrbracket$. This represents a term that is to be obtained by substituting t_2 for the first free variable in t_1 (and modifying the indices for the other free variables). However, the indices for the free variables in t_2 must be ‘bumped up’ by $(nl - nl')$ before this substitution is made. In the other systems, the needed increment to the indices of free variables is maintained explicitly with the term in the environment. Thus, the suspension term shown above would be represented, as it were, as $\llbracket t_1, 1, nl, (t_2, (nl - nl')) :: nil \rrbracket$; actually, ol and nl are needed in this term only for determining the adjustment to the free variables in t_1 with indices greater than 1, and devices for representing environments encapsulating such an adjustment simplify the actual notation used. The representation used in [ACCL90] and [Fie90] have the benefit of parsimony: no special syntax is required for environment terms and rules that are used for manipulating terms can also be used for manipulating terms in the environment. Notice, however, that the rule for moving substitutions under abstractions becomes more complex in that *every* term in the environment is now affected. Thus, from a term of the form $\llbracket (\lambda t_1), 1, nl, (t_2, (nl - nl')) :: nil \rrbracket$, this rule must produce a term that looks something like $(\lambda \llbracket t_1, 2, nl + 1, @1 :: (t_2, nl - nl' + 1) :: nil \rrbracket)$. In contrast, using our representation, this rule is required only to add a ‘dummy’ element to the environment and to make a *local* change to the embedding levels of the overall term. On a balance, the trade-offs in the two approaches appear to be even in the context of the overall rewriting systems. However, our representation seems to have an advantage if a simpler rewriting system, such as that obtained by eliminating the merging rules, is used.

In a different direction, the general idea of delaying substitutions appears to have been anticipated by de Bruijn in [Bru72] and [Bru78]. In the latter paper, de Bruijn actually presents a notation for lambda terms that includes mappings for transforming variable indices within terms. The specific notation presented in [Bru78] is quite cumbersome and, in addition, does not include any mechanisms for encoding the substitution operation needed for β -contraction. However, a special form of the general substitution operation that suffices for β -contraction has been described in the literature, and using laziness in its implementation results in a notation close to the one presented here. In particular, β -contraction is described in [Har89] by means of a binary function σ_n and a unary function τ_i^n on terms. These functions perform the following tasks: $\sigma_n(t_1, t_2)$ produces a term from t_1 by decreasing the indices for the $(n + 1)$ -st and later free variables by 1 and replacing the n -th free variable by t_2 after the indices for the free variables in t_2 have been ‘bumped up’ by n ; $\tau_i^n(t)$ produces the term that results from t by raising the indices for the i -th and later free variables in it by n . A similar set of functions is described by Staples in [Sta81]. Our notion of a suspension collapses these two functions into a common form and captures the effect of

(Clos)) can be eliminated is incorrect. However, as pointed out to us by P.-L. Curien, restricted versions of this rule and of other environment manipulating rules suffice from the perspective of simulating β -reduction in the notation presented there.

evaluating them in a delayed fashion. It is interesting to note that two indices *ol* and *nl* are needed in a term of the form $[[t, ol, nl, e]]$ to achieve this objective; an attempt to use only one index was made in [OS84] but could not be carried out to completion. We also observe that our notation actually generalizes the mentioned functions by allowing for environments that represent *multiple* non-dummy substitutions that are to be performed simultaneously.

The notation studied in this paper is intended to have practical utility. Our particular desire is that this notation serve as a substrate upon which coarser-grained representations for lambda terms may be developed that are eventually used in actual implementations. We explore this issue in a companion paper [Nad94]. One particular refinement we consider is that of eliminating the merging rules. These rules have a practical advantage in that it is only through them that substitution walks over the structure of a term can be combined. However, implementing these rules in their full generality can be cumbersome. Our approach to this is to capture some of their effects through auxiliary rules. The resulting rewrite system permits us to restrict our attention to only simple expressions. Another refinement consists of adding annotations to terms that determine whether or not they can be affected by substitutions generated by external β -contractions. We then use the refined notation to describe manipulations to lambda terms and to prove properties of such manipulations. It is this work that directly underlies the implementation that is being developed for λ Prolog [NJW93].

Acknowledgements

We are grateful to P.-L. Curien for his comments on an earlier version of this paper. Comments provided by John Hannan and an anonymous reviewer on an early presentation of the ideas in this paper in [NW90] helped in making us aware of related research. The first author also acknowledges the stimulus received from Mike O'Donnell and his students by their participation in a presentation of these ideas in Spring 1991. Work on this paper has been supported by NSF grants CCR-89-05825 and CCR-92-08465.

References

- [ACCL90] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, January 1990.
- [AP81] L. Aiello and G. Prini. An efficient interpreter for the lambda-calculus. *The Journal of Computer and System Sciences*, 23:383–425, 1981.
- [Bru72] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

- [Bru78] N. de Bruijn. Lambda-calculus notation with namefree formulas involving symbols that represent reference transforming mappings. *Indag. Math.*, 40:348–356, 1978.
- [Bru80] N. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.
- [CAB⁺86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
- [CCM87] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.
- [CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [Cur86a] P-L. Curien. Categorical combinators. *Information and Control*, 69:188–254, 1986.
- [Cur86b] P-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman, 1986.
- [Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
- [Fie90] John Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–15. ACM Press, January 1990.
- [Gal91] Jean H. Gallier. What’s so special about Kruskal’s theorem and the ordinal $\Gamma_0\Gamma$ A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53:199–260, 1991.
- [GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [Hal60] Paul R. Halmos. *Naive Set Theory*. D. Van Nostrand Company, Inc., 1960.
- [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.

- [HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
- [HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
- [HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.
- [Hue75] Gérard Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.
- [Hue86] Gérard Huet. Formal structures for computation and deduction. Unpublished course notes, Carnegie Mellon University, 1986.
- [KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
- [Klo80] J.W. Klop. *Combinatory Reduction Systems*. Mathematisch Centrum, Amsterdam, 1980.
- [Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem and Vázsonyi’s conjecture. *Trans. Amer. Math. Soc.*, 95:210–225, 1960.
- [Lev79] Azriel Levy. *Basic Set Theory*. Springer-Verlag, 1979.
- [MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.
- [Nad94] Gopalan Nadathur. A notation for lambda terms II: Refinements and applications. Technical Report CS-1994-01, Department of Computer Science, Duke University, January 1994.
- [NJW93] Gopalan Nadathur, Bharat Jayaraman, and Debra Sue Wilson. Implementation considerations for higher-order features in logic programming. Technical Report CS-1993-16, Department of Computer Science, Duke University, June 1993.
- [NM88] Gopalan Nadathur and Dale Miller. An overview of λ Prolog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

- [NW90] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, 1990.
- [OS84] Michael J. O’Donnell and Robert I. Strandh. Towards a fully parallel implementation of the lambda calculus. Technical Report JHU/EECS-84/13, Johns Hopkins University, 1984.
- [Pau87] Lawrence R. Paulson. The representation of logics in higher-order logic. Technical Report Number 113, University of Cambridge, Computer Laboratory, August 1987.
- [Pau88] L. Paulson. The foundations of a generic theorem prover. Technical Report Number 130, University of Cambridge, Computer Laboratory, March 1988.
- [PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
- [Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.
- [Sta81] John Staples. A new technique for analysing parameter passing, applied to the lambda calculus. *Australian Computer Science Communications*, 3(1):201–210, May 1981.
- [Yok89] Hirofumi Yokouchi. Church-Rosser Theorem for a rewriting system on categorical combinators. *Theoretical Computer Science*, 65:271–290, 1989.