# A Correctness Proof for Combinator Reduction with Cycles

WILLIAM M. FARMER, JOHN D. RAMSDELL, and RONALD J. WATRO
The MITRE Corporation

Turner popularized a technique of Wadsworth in which a cyclic graph rewriting rule is used to implement reduction of the fixed point combinator $Y$. We examine the theoretical foundation of this approach. Previous work has concentrated on proving that graph methods are, in a certain sense, sound and complete implementations of term methods. This work is inapplicable to the cyclic $Y$ rule, which is unsound in this sense since graph normal forms can exist without corresponding term normal forms. We define and prove the correctness of combinator head reduction using the cyclic $Y$ rule; the correctness of normal reduction is an immediate consequence. Our proof avoids the use of infinite trees to explain cyclic graphs. Instead, we show how to consider reduction with cycles as an optimization of reduction without cycles.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.2 [**Programming Languages**]: Language Classification—*applicative languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Theory

Additional Key Words and Phrases: Combinators, correctness proof, functional programming, graph reduction, term rewriting systems

## 1. INTRODUCTION

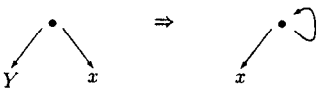
Combinatory logic was developed independently by Schönfinkel [13] and Curry [3] as a foundation for mathematics and logic. More recently, combinatory logic has gained relevance in computer science. In particular, one approach to implementing functional programming languages is to compile functional programs into combinator terms which are then executed via combinator reduction (see Peyton Jones [11] for a good reference on using combinators to implement functional programming languages.)

Combinator reduction is usually implemented using directed graphs to represent combinator terms. The following optimizations are widely used in the reduction process:

(1) Terms are represented by directed acyclic graphs (DAGs) so that multiple occurrences of a subterm can be represented by a single graph structure.

This work was supported by the MITRE Sponsored Research Program.
Authors' address: The MITRE Corporation, Burlington Road, Bedford, MA 01730.

(2) The reduction rule $Yx = x(Yx)$ for the fixed point combinator $Y$ is applied using a method popularized by Turner [16] that introduces directed cycles in graphs, as indicated by the following diagram:




(3) The argument stack is stored in the graph using the pointer reversal technique that originated in the garbage collection algorithm of Schorr and Waite [14].

This paper gives a correctness proof for the second technique. There are several areas of existing work. The first optimization above, DAG reduction, has been completely justified using graph rewriting by Staples [15] and by Barendregt et al. [2]. The second optimization, the cyclic $Y$ rule, has been studied from a number of different viewpoints. In Barendregt et al. [2], the issue of the cyclic $Y$ rule is discussed, and some key ideas are elucidated, but the basic correctness property is not proved.

There are both semantic and syntactic justifications for the $Y$ rule. On the semantic side, Diller [5] interprets $Y$ as the least fixed point operator inside lambda calculus and provides a justification for the cyclic rule in this setting. A second semantic approach to the $Y$-correctness result is due to Felleisen [7], who constructs a calculus extending lambda calculus and including constructs for state and control. In this extended calculus, Felleisen defines a combinator $Y!$, an imperative cyclic version of $Y$, and proves that $Y!$ satisfies an operational fixed-point property.

We prefer a syntactic approach to the $Y$ rule, because such an approach is more general, not requiring specific assumptions about models, and more elementary, avoiding the machinery of denotational semantics. One interesting syntactic approach involves an infinite sequence of term rewritings, a topic that has recently been studied by Dershowitz and Kaplan [4]. The relationship between graph rewriting and infinite term rewriting is being pursued by Farmer and Watro [6].

In this paper, we develop a different approach, one based on the intuition that the cyclic rule is merely an optimization of the ordinary rule. Our proof utilizes a modification of cyclic reduction in which additional information is maintained in order to resolve cycles. The process is proved correct using the additional information, and the correctness of ordinary cyclic reduction follows immediately.

The correctness of the Schorr–Waite algorithm has been the subject of several formal analyses (see Morris [10]). Let us remark here concerning one interesting interaction between graphs with cycles and Schorr–Waite pointer reversal. Head reduction on a graph requires a search down the left-most branch of the graph to locate the head symbol (if any), and then a search back up to accumulate the appropriate number of arguments. This down-and-then-up search procedure is conveniently implemented by reversing the direction of a link after it is traversed. When a graph contains a cycle on the left-most path, the pointer-reversal algorithm will have its search for the head symbol redirected toward the root of the graph. This redirected search will terminate at a symbol representing the

bottom of the stack. Part of our work in this paper is to analyze the situation in which the left-most path of a graph is cyclic. This work can be used to show that if pointer reversal produces the stack bottom as the head symbol, then the original term being reduced has no head normal form.

We consider primarily head reduction and head normal forms for combinator expressions. This focus originates from work on a formal model of the Curry Chip [12], a combinator reduction machine implemented in VLSI. The Curry Chip uses the standard optimizations in reducing combinators to head normal form. While studying the Curry Chip, we recognized the need for a proof of the correctness of cyclic head reduction. The correctness of normal order reduction follows easily from the correctness of head reduction.

## 2. BACKGROUND

We define a *combinatory rewriting system* to be a special type of term rewriting system. The terms in a combinatory rewriting system are constructed from variables, constants called basic combinators, and a single binary operation called application. It is traditional to denote application by juxtaposition and to assume left associativity as the default for terms. The rules in a combinatory rewriting system consist of one equation of the form $Cx_1 \cdots x_n = t(x_1, \ldots, x_n)$ for each basic combinator $C$. Here $x_1, \ldots, x_n$ are distinct and $t(x_1, \ldots, x_n)$ is a term containing no variables other than $x_1, \ldots, x_n$.

The rewriting system with basic combinators $S$ and $K$ and rules $Sxyz = xz(yz)$ and $Kxy = x$ is a well-known example of a combinatory rewriting system. This system is complete in the sense that any combinator is representable in terms of $S$ and $K$. For example, the combinator $I$ with rule $Ix = x$ can be represented by $SKK$, since $SKKx$ rewrites to $Kx(Kx)$ using the $S$ rule, and this result rewrites to $x$ using the $K$ rule. Combinatory rewriting systems designed for implementation purposes often contain additional basic combinators beyond $S$ and $K$; in particular, an explicit fixed-point combinator $Y$ is crucial for efficient implementation of recursion.

Let the *head symbol* of a term be the leading variable or basic combinator appearing in the term. If a term $t$ rewrites to $t'$ in one step, we write $t \rightarrow t'$. The reflexive and transitive closure of $\rightarrow$ is $\rightarrow^*$. A term is a normal form if none of the rewrite rules apply to it. If $t \rightarrow^* t'$, and $t'$ is a normal form, then $t'$ is said to be a normal form of $t$. Combinatory rewriting systems are left linear and nonambiguous and hence confluent (see [8] or [9]); thus a term has at most one normal form. A term is a head normal form if it is equal to $Ht_1 \cdots t_n$, where $H$ is either a variable or a basic combinator whose axiom requires more than $n$ arguments. A term can have more than one head normal form, but the head symbol and the number of arguments in a head normal form are unique.

### 2.1 Graph Terms

Assume some fixed combinatory rewriting system. In this subsection, we formalize the notion of a graph term as a set of simple equations, and in the next subsection we define head reduction for graph terms. A graph term is meant to be a generalization of a parse tree. We define it as a set of equations to simplify

our proofs. The construction of a directed, rooted graph from a graph term is described below.

*Definition* 1. An equation is *simple* if it has the form $x = yz$ (called an *application equation*), the form $x = y$ (called a *transfer equation*), or the form $x = C$ (called a *constant equation*), where $x$, $y$, and $z$ are variables and $C$ is a constant.

*Definition* 2. A *graph term* is a pair $G = (E, r)$ where $r$ is a variable called the root variable of $G$ and $E$ is a finite set of simple equations such that no variable occurs more than once as the left side of an equation in $E$. A *leaf variable* of $G$ is a variable which occurs in the right side of an equation, but not in the left side of an equation, in $E$. A *path* in a graph term $G$ is a finite or infinite sequence of equations $\{e_i\}$ contained in $G$ such that the left-side variable of $e_{i+1}$ always occurs on the right side of $e_i$. A graph term is *acyclic* if all paths in it are finite. The *head path* of $G$ is the maximal path $\{e_i\}$ such that $r$ is the left side of $e_1$, and such that the left side of $e_{i+1}$ is the first symbol on the right side of $e_i$. If the head path of $G$ is finite, then the *head symbol* of $G$ is the first symbol on the right side of the last equation in the head path of $G$.

A graph term $G$ corresponds to a graph in the following manner. The variables in $G$ correspond to the nodes of the graph. An application equation defines a directed edge from the left-hand variable to each of the right-hand variables. A transfer equation defines a single directed edge from the left-hand variable to the right-hand variable. A constant equation labels a leaf node of the graph with a constant. The leaf variables of $G$ correspond to leaf nodes of the graph which are not labeled by constants.

We now show how an acyclic graph term represents a term. Let $G$ be an acyclic graph term with root variable $r$. The *value* of a term $t$ in $G$, written $\mathrm{val}(t, G)$, is defined inductively as follows:

(1) If $t$ is a constant or a leaf variable of $G$, then $\mathrm{val}(t, G) = t$.
(2) If $t$ is a nonleaf variable of $G$, then $\mathrm{val}(t, G) = \mathrm{val}(s, G)$, where $t = s$ is in $G$.
(3) If $t = s(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ includes all the variables in $t$, then $\mathrm{val}(t, G) = s(\mathrm{val}(x_1, G), \ldots, \mathrm{val}(x_n, G))$.

Since $G$ is acyclic, $\mathrm{val}(t, G)$ is well defined for all terms in $t$. The term that $G$ represents is the value of $r$ in $G$.

Acyclic graph terms include the parse trees and DAGs constructed from ordinary terms. Cyclic graph terms include structures containing directed cycles such as rational trees. For example, the graph term

$$G = (\{x = yx, y = I\}, x)$$

could be imagined as representing the infinite term

$$I(I(I( \cdots .$$

Given an arbitrary term $t$, let $G(t)$ be any acyclic graph term which represents $t$. For example, given a one-to-one mapping from terms $t$ to variables $x_t$, $G(t)$ for a nonvariable term $t$ could be the graph term $(E(t), x_t)$, where $E(t)$ is defined

inductively as follows:

(1) If $t = C$, then $E(t) = \{x_t = C\}$, where $C$ is a constant.
(2) If $t = v$, then $E(t) = \varnothing$, where $v$ is a variable.
(3) If $t = s_1 s_2$, then $E(t) = \{x_t = x_{s_1} x_{s_2}\} \cup E(s_1) \cup E(s_2)$.

This graph term represents the DAG that is formed from the parse tree of $t$ by eliminating redundant subexpressions. Notice that there are no transfer equations in $E(t)$; transfer equations are ordinarily introduced in the act of reducing a graph term.

### 2.2 Head Reduction of Graph Terms

An acyclic graph term can be reduced in essentially the same manner as a term. We only define the *head reduction* of an acyclic graph term. Suppose that $G = (E, r)$ is an acyclic graph term. Head reduction begins with a search to locate the head symbol of $G$. We generate two finite sequences of variables $x_i$ and $y_j$ in $G$. The $x_i$ sequence corresponds to the head path of $G$, and some final segment of the $y_j$ sequence comprises the arguments of the head symbol. Due to the possible presence of transfer equations, the $y_j$ sequence may be strictly shorter than the $x_i$ sequence.

The $x_i$ and $y_j$ sequences are defined inductively as follows. Set $x_0 = r$. Assume that $x_i$ is defined for all $i$ such that $0 \leq i \leq p$ and that $y_j$ is defined for all $j$ such that $1 \leq j \leq q$ where $p, q \geq 0$. There are three cases to consider:

(1) If $x_p = C$ is in $E$ or $x_p$ is a leaf variable of $G$, then the $x_i$ and $y_j$ sequences are complete.
(2) If $x_p = z$ is in $E$, then set $x_{p+1} = z$.
(3) If $x_p = zw$ is in $E$, then set $x_{p+1} = z$ and $y_{q+1} = w$.

For an acyclic graph term, this construction always terminates. If $x_p$ is a leaf variable of $G$, then the term represented by $G$ is in head normal form, so $G$ is defined not to head reduce. Let $m$ and $n$ be the lengths of the completed $x_i$ and $y_j$ sequences, respectively, and assume that the construction terminates with the constant $C$. Suppose that the axiom for $C$ requires $k$ arguments. If $k > n$, then the term represented by $G$ is in head normal form, so again $G$ is defined not to head reduce. If $k \leq n$, then $G$ head reduces as described below.

To define the head reduction of $G$, note that $E$ contains the equations

$$\begin{aligned} x_m &= C \\ x_{m-1} &= x_m t_m \\ &\vdots \\ x_i &= x_{i+1} t_{i+1} \\ &\vdots \\ x_0 &= x_1 t_1 \end{aligned}$$

where each $t_i$ is either an element of the $y_j$ sequence or is empty. Use the equation for $x_m$ to eliminate the first occurrence of the variable $x_m$ in the equation for

$x_{m-1}$. Continue in this fashion, using the new equation for $x_{m-1}$ to eliminate the first occurrence of $x_{m-1}$ in the equation for $x_{m-2}$. Because $k \leq n$, this procedure will produce an equation of the form

$$x_h = Cy_n y_{n-1} \cdots y_{n-k+1}.$$

We apply the reduction rule for $C$ to the right side of this equation, producing $x_h = t$ where $t$ may contain the variables $y_n$ through $y_{n-k+1}$.

We are finally ready to define $G'$, the graph term that results from performing one head reduction step on $G$. Our goal is to incorporate the effect of the equation $x_h = t$ into $G'$. Form $E^-$ from $E$ by removing the equation beginning with $x_h$. Let $F$ be any nonempty set of simple equations such that (1) $(F, x_h)$ is an (acyclic) graph term representing $t$ and (2) $z$ occurs in both $E^-$ and $F$ only if $z = x_h$ or $z$ occurs in $t$. Then $G' = (E^- \cup F, r)$. It is easy to check that $G'$ is a graph term. Note that $G'$ has the same root variable as $G$.

For example, if $C = S$, then $k = 3$ and $t = y_n y_{n-2}(y_{n-1} y_{n-2})$. Hence we may define $F$ to be

$$\{x_h = w_1 w_2,\ w_1 = y_n y_{n-2},\ w_2 = y_{n-1} y_{n-2}\}$$

provided $w_1$ and $w_2$ do not occur in $E^-$.

*Remark.* It is possible that there is a variable $x$ occurring in $G$ and $G'$ such that there is a path from $r$ to $x$ in $G$ but no such path in $G'$. Variables such as $x$ will often be ignored in graph terms because they are not needed in determining what a graph term represents or in head reduction of a graph term.

The close relation between head reduction of $t$ and head reduction of $G(t)$ is fully analyzed by Barendregt et al. [2]. We provide here just the statements of the key results. It is easy to check that if acyclic graph head reduction of $G(t)$ terminates with a graph term $G$, then $G$ represents a head normal form of $t$. To verify that graph head reduction always terminates whenever the term has a head normal form, one uses the following result.

*Definition* 3. A finite reduction sequence is a *quasi-head* reduction sequence, written $s \rightarrow^q t$, if it contains at least one head reduction. An infinite reduction sequence is quasi-head if it contains infinitely many head reduction steps.

THEOREM 1. *If there is an infinite quasi-head reduction starting from t, then t has no head normal form.*

See [1, exercise 13.6.13] for a proof of the theorem. This result implies that head reduction of $t$ terminates if and only if acyclic graph head reduction of $G(t)$ terminates and ensures that acyclic reduction is a sound and complete implementation (in the sense of [2]) of a combinatory rewriting system.

## 3. CYCLIC COMBINATOR REDUCTION

The fixed-point axiom $Yx = x(Yx)$ is unusual in that it creates a new instance of the subterm that it rewrites. Recall that one key advantage of graph reduction is that one graph step can reduce several occurrences of the same redex in a term. The cyclic $Y$ rule has the advantage that one reduction replaces possibly infinitely many future reductions.

Cyclic reduction proceeds exactly as the acyclic reduction defined in the last section, except that it is possible for a cyclic graph term to have no head symbol and that we will implement the $Y$ rule differently. $Y$ reduction will now replace the equation $x = Yy$ with the equation $x = yx$. This creates a loop edge in the graph, and further reduction can create directed cycles of arbitrary length. While a DAG always corresponds in an obvious way to a (finite) term, cyclic graphs do not. A variable need not have a finite value in a cyclic graph term. This makes a substantial difference in correctness proofs.

*Example* 1. Consider the term $YI$, where $I$ is the identity combinator with the axiom $Ix = x$. Ordinary term rewriting produces an infinite reduction sequence

$$YI \rightarrow I(YI) \rightarrow YI \rightarrow \cdots.$$

Graph rewriting, as defined in this section, cannot proceed past two steps, producing the graph term $G = (\{x = x\}, x)$.


The graph term $G$ has no head symbol, and hence it is a head normal form with respect to graph head reduction. Another unusual feature of the graph term $G$ is that it does not represent any finite or infinite binary application tree. One can view $G$ as representing a nonterminating calculation, based on the fact that $G$ contains a single transfer equation that sends the root back to itself. This type of graph term is avoided when graph rewriting is defined as in [2], because the graph term $(\{x = zx, z = I\}, x)$ reduces to itself instead of $G$. This approach generates an infinite graph reduction sequence starting from $YI$. Both definitions of graph rewriting have natural motivations unrelated to this example; in the context of this example, our definition provides an additional way to recognize nontermination.

*Remark.* We have presented our reduction algorithm using explicit transfer equations and without mechanisms for eliminating transfer equations in the reduction process. It is also possible to view transfer equations as a special type of application equation, and to eliminate some transfers either in the reduction process or in garbage collection. To accomplish this, a special constant symbol $I$ is used, and the application equation $x = Iy$ represents the transfer equation $x = y$. The symbol $I$ is treated partly as a combinator, in that $Ixy = xy$ can be applied as a reduction rule, and partly as a transfer marker, in that $It$ reduces to $It'$ if $t$ reduces to $t'$.

### 3.1 Augmented Cyclic Reduction

In order to prove correctness, we define an augmentation of the cyclic reduction process that maintains additional information in a graph term consisting of (1) superscripts attached to certain variables and (2) additional equations. The additional information is generated each time a $Y$ rule is applied. A graph term $G = (E, r)$ with this added information is called an *augmented graph term.* The set $E$ of equations is partitioned into two disjoint pieces: $E_1$ contains the equations

from the basic algorithm, while $E_2$ contains the additional equations (which need not be simple).

The additional equations appearing in $E_2$ will resolve any cycles among the equations in $E_1$. That is, we can define the value of a term $t$ in $G$ to be a unique (finite) term analogously to how we defined the value of a term in an acyclic graph term.

Augmented cyclic head reduction proceeds in the following manner. Initially, $G$ is an acyclic graph term representing the term we want to reduce with $E_2$ empty. A step in the augmented process works as follows. For combinators other than $Y$, proceed as usual, making use of just the equations in $E_1$. Suppose that the $Y$ rule is applied to an equation $x = Yz$ derived from the set $E$. The new set $E'$ of equations is constructed from $E$ by (1) removing from $E_1$ the equation beginning with $x$; (2) adding to $E_1$ the equation $x = zx^i$, where $i$ is the first positive integer that has not already been used as a superscript on the variable $x$; and (3) adding to $E_2$ the equation $x^i = t$, where $t$ is the value of $x$ in $G$.

If a term $t$ is given and the augmented cyclic reduction algorithm is applied to $G(t)$, producing a sequence of (augmented) graph terms $G_1, \ldots, G_i, \ldots$, then the result of applying the ordinary cyclic reduction algorithm to $t$ is just $G_1', \ldots, G_i', \ldots$, where $G_i'$ is obtained form $G_i$ by (1) removing the $E_2$ equations from $G_i$ and (2) erasing the superscripts from all the variables in $G_i$. It follows then that if the augmented algorithm always terminates when $t$ has a head normal form, then so does the ordinary algorithm.

This notion of augmented cyclic head reduction is best explained with an example.

*Example* 2. Consider the term $YY$. As with the term $YI$ in Example 1, ordinary term rewriting produces an infinite reduction sequence, but graph rewriting cannot proceed past two steps.


Notice that the last graph term has the form $(\{x = xx\}, x)$. Our augmentation of graph rewriting provides enough information to convert this last graph into a term and to recognize that a cycle has occurred in the term reduction process.

The augmented algorithm begins with the simple equations that define the term $YY$:

$$\begin{array}{l} x = yz \\ y = Y \\ z = Y \end{array} \quad \Big|$$

The first graph reduction step produces

$$\begin{array}{l} x = zx^1 \\ y = Y \\ z = Y \end{array} \quad \Big| \quad x^1 = YY$$

The second graph reduction step produces

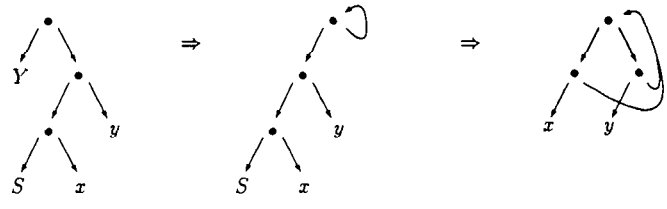
$$\begin{array}{l|l} x = x^1 x^2 & x^1 = YY \\ y = Y & x^2 = Y(YY) \\ z = Y & \end{array}$$

Reduction terminates at this point, because there is no head symbol in the last graph. The term represented by this last graph is determined by solving for $x$ using the equations in $E_1 \cup E_2$. This gives the result $x = YY(Y(YY))$, and this term appears in the term head reduction of $YY$. Since $YY$ is a prefix of $YY(Y(YY))$, it follows that term head reduction of $YY$ is infinite:

$$YY \rightarrow^* YY(Y(YY)) \rightarrow^* (YY(Y(YY)))(Y(YY)) \cdots.$$

In the general case, the term we construct as the value of the last graph is either a head normal form of the original term, or is part of the proof that the original term has no head normal form.

In the two examples above, all cycles that occur have length one. Cycles of arbitrary size can also be produced. For example, the term $Y(Sxy)$ reduces as follows:




## 3.2 Correctness Proof

Let $G$ be an augmented graph term produced by the augmented algorithm. For each variable in $G$, there is a unique term that is the value of that variable in $G$. The value of a superscripted variable is given directly by an equation in $G$, and the value of an ordinary variable $x$ is computed starting from the simple equation with $x$ as its left side. In this section the value of $x$ in $G$ is denoted by $x_G$, and we drop the subscript when only one graph is under discussion.

We need one additional piece of notation: $x \rightarrow^{=q} y$ means that either $x = y$ or $x \rightarrow^q y$.

LEMMA 1

(a) *In any run of the augmented cyclic graph head reduction algorithm, if $x$ and $x^i$ are variables in $G$, then $x^i \rightarrow^q x$.*

(b) *If $G'$ is produced from $G$ by one step of augmented cyclic graph head reduction and $y$ is a variable in both $G$ and $G'$, then $y_G \rightarrow^* y_{G'}$.*

PROOF. We prove parts (a) and (b) by a joint induction. Consider some run of the augmented algorithm. The initial graph contains no superscripted variables, so (a) holds for the first graph. Let $G = (E, r)$ be any augmented graph term for which (a) holds; we prove that (b) holds for the reduction step from $G$

to $G'$ as follows: In $E_1$, we must have a finite chain of equations:

$$\begin{aligned} x_0 &= x_1^* t_1 \\ x_1 &= x_2^* t_2 \\ &\vdots \\ x_i &= x_{i+1}^* t_{i+1} \\ &\vdots \\ x_m &= C. \end{aligned}$$

Here $x_i^*$ denotes either $x_i$ itself or $x_i^j$ for some positive superscript $j$, and $t_i$ is either empty or some variable $y$. Because (a) is true for $G$, we have that

$$x_h \rightarrow^{=q} Cy_n \cdots y_{n-k+1}$$

following the same approach as used in Section 2.2. The construction of $G'$ introduces a new equation for $x_h$ by applying the $C$ rule. All other equations of $G$ are carried over unchanged into $G'$. It follows that $x_{hG} \rightarrow^q x_{hG'}$. An arbitrary variable appearing in $G$ will have a new value in $G'$ only if it depends somehow on $x_h$; hence, $y_G \rightarrow^* y_{G'}$.

To complete the proof of the lemma, we must prove that (a) holds for $G'$. Let $x^i$ be an arbitrary superscripted variable in $G'$. If $x^i$ is not in $G$, then the equation for $x^i$ was created by applying the $Y$ rule to $G$, and (a) clearly holds in this case. If $x^i$ is in $G$, then $x_G^i \rightarrow^q x_G$ since (a) holds for $G$, and $x_G \rightarrow^* x_{G'}$ since (b) holds for the reduction step from $G$ to $G'$. Since $x_G^i = x_{G'}^i$, we obtain that $x_{G'}^i \rightarrow^q x_{G'}$, and thus (a) holds for $G'$. □

LEMMA 2. *Assume that $r$ is the root variable of an acyclic graph term $G_1$. If $G_1, \ldots, G_i, \ldots$ is a sequence of augmented graph terms produced by augmented cyclic head reduction, then $r_{G_1}, \ldots, r_{G_i}, \ldots$ is a quasi-head reduction sequence of terms.*

PROOF. Each time a graph reduction step occurs, there is a unique variable $x_h$ where the reduction process applies. The value of $x_h$ is head reduced for one step by the graph reduction process, and $r \rightarrow^{=q} x_h y_h \cdots y_1$. It follows that $r_{G^i} \rightarrow^q r_{G^{i+1}}$. □

LEMMA 3. *If cyclic head reduction of $G(t)$ produces a graph without a head, then there is an infinite quasi-head reduction starting at $t$.*

PROOF. Assume that the augmented graph head reduction is performed starting from a graph term $G = (E, r)$, and that an augmented graph term $G' = (E', r)$ is produced in which the head search process enters a cycle. This means

that in $E_1'$ we have a chain of equations of the form

$$\begin{aligned} x_0 &= x_1^* t_1 \\ x_1 &= x_2^* t_2 \\ &\vdots \\ x_i &= x_{i+1}^* t_{i+1} \\ &\vdots \\ x_m &= x_i^* t_{n+1}. \end{aligned}$$

It follows using Lemma 1(a) that

$$x_i \rightarrow^{=q} x_i^* y_{n+1} \cdots y_{i+1}.$$

If $x_i^* = x_i^j$, then we have that $x_i \rightarrow^{=q} x_i^j y_{n+1} \cdots y_{i+1}$, and combining this with $x_i^j \rightarrow^q x_i$ yields that there is an infinite quasi-head reduction starting at $x_i$. If $x_i^* = x_i$, then there must exist a $k$ such that $i < k \leq m$ and $x_k^* = x_k^j$ for some $j$, because otherwise the cycle would be present in the augmented set of equations, which is impossible. This gives that $x_i \rightarrow^q x_i y_{n+1} \cdots y_{i+1}$, and again there must be an infinite quasi-head reduction starting at $x_i$. Because $r_G \rightarrow^q r_{G'}$ and $r \rightarrow^{=q} x_i y_i \cdots y_1$, we have an infinite quasi-head reduction starting at $t$, as required. □

THEOREM 2. *Cyclic combinator head reduction is correct, meaning that*

(*a*) *If a term t has a head normal form, then cyclic reduction of G(t) finds the head symbol and the number of arguments in a head normal form.*
(*b*) *If a term t has no head normal form, then cyclic reduction of G(t) either produces an infinite sequence of graph terms or produces a graph term with no head symbol.*

PROOF. Consider augmented cyclic head reduction applied to $G(t) = (E, r)$. Ordinary cyclic reduction can be extracted by considering just the $E_1$ sets and ignoring the superscripts. There are three possibilities: If the reduction terminates in a graph term with a head symbol, then the head symbol and number of arguments are determined from the final $E_1$ without superscripts, and Lemma 2 assures that they are correct. If the reduction terminates in a graph term without a head, then Lemma 3 assures that $t$ has no head normal form. If the reduction is infinite, Lemma 2 and Theorem 1 establish that $t$ has no head normal form. □

The augmented cyclic reduction process can provide actual finite terms for the arguments in a head normal form, while ordinary cyclic head reduction provides only possibly cyclic graph terms as arguments. When fully normal forms are required, head reduction is recursively applied to the arguments in a head normal form.

## REFERENCES


1. BARENDREGT, H. P. *The Lambda Calculus, Its Syntax and Semantics*, rev. ed. North-Holland, New York, 1984.
2. BARENDREGT, H. P., VAN EEKELEN, M. C. J. D., GLAUERT, J. R. W., KENNAWAY, J. R., PLASMEIJER, M. J., AND SLEEP, M. R. Term graph rewriting. In *PARLE—Parallel Architectures and Languages Europe. Springer Lecture Notes in Computer Science,* vol. 259. Springer, New York, 1987, pp. 141–158.
3. CURRY, H. B. Grundlagen der kombinatorischen Logik. *Amer. J. Math. 52,* 1930.
4. DERSHOWITZ, N., AND KAPLAN, S. Rewrite, Rewrite, Rewrite .... In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1989, pp. 250–259.
5. DILLER, A. *Compiling Functional Languages.* Wiley, New York, 1988.
6. FARMER, W. M., AND WATRO, R. J. Redex Capturing in Term Graph Rewriting. Tech. Rep. M89-36, The MITRE Corporation, Bedford, Mass., July 1989.
7. FELLEISEN, M. The calculi of lambda-$\nu$-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages. Ph.D. dissertation, Indiana Univ., 1987.
8. HUET, G. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM 27,* 4 (Oct. 1980), 797–821.
9. KLOP, J. W. *Combinatory Reduction Systems.* Mathematisch Centrum, Amsterdam, 1980.
10. MORRIS, J. M. A proof of the Schorr–Waite algorithm. In *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt, Eds. NATO Advanced Study Institute, D. Reidel, 1982, pp. 43–51.
11. PEYTON JONES, S. L. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J., 1987.
12. RAMSDELL, J. D. The CURRY chip. In *1986 ACM Symposium on LISP and Functional Programming* (Cambridge, Mass., Aug. 1986), pp. 122–131.
13. SCHÖNFINKEL, M. Über die bausteine der mathematischen Logik. *Math. Ann. 92* (1924), 305–316.
14. SCHORR, H., AND WAITE, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM 10,* 8 (Aug. 1967), 501–506.
15. STAPLES, J. Computation on graph-like expressions. *Theoretical Comput. Sci. 10* (1980), 171–185.
16. TURNER, D. A. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9* (1979), 31–49.
17. WADSWORTH, C. P. Semantics and Pragmatics of the Lambda Calculus. Ph.D. dissertation, Programming Research Group, Oxford Univ., Oxford, U.K., 1971.


Received November 1988; revised July 1989; accepted August 1989