Explicit Cyclic Substitutions

Kristoffer Høgsbro Rose

DIKU, University of Copenhagen,
Universitetsparken 1, 2100 København Ø, Denmark
Internet: kris@diku.dk

DIKU Semantics Note D–143, February 8, 1993

Abstract. In this paper we consider rewrite systems that describe the λ-calculus enriched with recursive and non-recursive local definitions by generalizing the 'explicit substitutions' used by Abadi, Cardelli, Curien, and Lévy [1] to describe sharing in λ-terms. This leads to 'explicit cyclic substitutions' that can describe the mutual sharing of local recursive definitions. We demonstrate how this may be used to describe standard binding constructions (**let** and **letrec**)—directly using substitution and fixed point induction as well as using 'small-step' rewriting semantics where substitution is interleaved with the mechanics of the following β-reductions.

With this we hope to contribute to the synthesis of denotational and operational specifications of sharing and recursion.

Contents.

1 Introduction

Most functional programming languages include some form of 'local binding' that makes sharing explicit, *e.g.*, the expression $(2^3+3^2)\times(2^3-3^2)$ can be computed by the following expression in a generic functional language with explicit sharing of the subexpressions 2^3 and 3^2:

$$\textbf{let } a = 2^3 \textbf{ and } b = 3^2 \textbf{ in } (a+b)\times(a-b)$$

It is clear that the two expressions have the same denotation since they compute the same value. But if we wish to give the two expressions a *rewrite semantics* in the form of a rewrite system describing how to obtain this value *operationally*,

Published as [23] supported by the Danish Research Council (SNF) project DART.

then the sharing matters because the two expressions rewrite in a different number of steps due to the duplication of computation in the first. The unshared version may rewrite in the following seven steps (where we underline the next subexpression to be reduced):

$$\begin{array}{rl}
(\underline{2^3} + 3^2) \times (2^3 - 3^2) & \Rightarrow (8 + \underline{3^2}) \times (2^3 - 3^2) \\
& \Rightarrow \underline{(8+9)} \times (2^3 - 3^2) \\
& \Rightarrow 17 \times (\underline{2^3} - 3^2) \\
& \Rightarrow 17 \times (8 - \underline{3^2}) \\
& \Rightarrow 17 \times \underline{(8-9)} \\
& \Rightarrow \underline{17 \times (-1)} \\
& \Rightarrow -17
\end{array}$$

But the shared version rewrites in five steps:

$$\begin{array}{l}
\textbf{let } a = \underline{2^3} \textbf{ and } b = 3^2 \textbf{ in } (a+b) \times (a-b) \\
\quad \Rightarrow \textbf{let } a = 8 \textbf{ and } b = \underline{3^2} \textbf{ in } (a+b) \times (a-b) \\
\quad \Rightarrow \textbf{let } a = 8 \textbf{ and } b = 9 \textbf{ in } \underline{(a+b)} \times (a-b) \\
\quad \Rightarrow \textbf{let } a = 8 \textbf{ and } b = 9 \textbf{ in } 17 \times \underline{(a-b)} \\
\quad \Rightarrow \textbf{let } a = 8 \textbf{ and } b = 9 \textbf{ in } \underline{17 \times (-1)} \\
\quad \Rightarrow -17
\end{array}$$

The situation gets even more severe for *cyclic sharing*, *e.g.*, the following expression computes the parity of 37:

$$\begin{array}{lll}
\textbf{letrec} & \mathit{even}(n) & = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } \mathit{odd}(n-1) \\
\textbf{and} & \mathit{odd}(n) & = \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } \mathit{even}(n-1) \\
\multicolumn{3}{l}{\textbf{in } \mathit{odd}(37)}
\end{array}$$

The only non-recursive representation of this is the complete 'unfolding' of the problem into an infinite expression like the one shown in the box below.

$$\boxed{\begin{array}{lll}
\textbf{let} & \mathit{even}(n) = & \textbf{if } n = 0 \textbf{ then } 1 \\
& & \textbf{else let} \;\; \mathit{even}(n) = \textbf{if } n = 0 \textbf{ then } 1 \\
& & \qquad\qquad\qquad\qquad \textbf{else let} \ldots \\
& & \qquad\quad \textbf{and } \mathit{odd}(n) \;\; = \textbf{if } n = 0 \textbf{ then } 0 \\
& & \qquad\qquad\qquad\qquad \textbf{else let} \ldots \\
& & \qquad\quad \textbf{in } \mathit{odd}(n-1) \\
\textbf{and} & \mathit{odd}(n) \;\; = & \textbf{if } n = 0 \textbf{ then } 0 \\
& & \textbf{else let} \;\; \mathit{even}(n) = \textbf{if } n = 0 \textbf{ then } 1 \\
& & \qquad\qquad\qquad\qquad \textbf{else let} \ldots \\
& & \qquad\quad \textbf{and } \mathit{odd}(n) \;\; = \textbf{if } n = 0 \textbf{ then } 0 \\
& & \qquad\qquad\qquad\qquad \textbf{else let} \ldots \\
& & \qquad\quad \textbf{in } \mathit{even}(n-1) \\
\multicolumn{3}{l}{\textbf{in } \mathit{odd}(37)}
\end{array}}$$

The traditional way to handle such infinite specifications is by least fixed point induction with which it is quite easy to see that the parity of 37 can be computed by unfolding 'just' 37 times—this is interesting because 37 is, after all, much less than infinity! But from a rewriting point of view this kind of argument is unsatisfactory because the unfolding clearly does not preserve sharing: we effectively *copy* the repeated piece of code an unlimited number of times—thus it is not possible to see what parts of the resulting term stems from the same piece of the original. Again—this may be a very useful property in denotational semantics where the goal is to identify algorithms with the same denotation but if we wish to study the rewrite system then we need insight into the mechanics of unfolding.

Thus we feel that there is a need for describing the relation between sharing in the forms discussed above and traditional ways of describing the meaning of such expressions. We will investigate this in the archetypical setting of 'enriched λ-calculi:' we will describe how the λ-calculus extended with explicit notation for the two kinds of sharing is related to the traditional λ-calculus where sharing is described implicitly using substitution and fixed point combinators.

Relation to other work

This work grew out of work in my thesis [24, part I] on generalizing the "Explicit Substitutions" rewrite system of Abadi, Cardelli, Curien, and Lévy [1] to handle cyclic structures. The chosen approach is similar to the "enriched λ-calculus" used to describe and implement functional programming languages by, *e.g.*, Peyton Jones [20], and to the "λ_{B}-calculus" graph reduction system of Ariola and Arvind [3] in that they intend to support mutual recursion directly. However, it differs in that there is no implied underlying 'execution model:' the description presented here is a pure, self-contained rewriting system. The work described here has been inspired by the 'substitution pushing' used by the "Call-by-Mix" strategy of Grue [11]. The current trend of 'optimal λ-reduction' also touches on sharing although only as a method to minimise the number of reductions [18, 17, 10, 19]

In fact a lot of the work done within **term graph rewriting** [6] seems to be aiming at achieving the same results as we are: to provide a convenient yet formal specification method that makes it possible to reason about rather than abstract away from sharing and recursion; *cf.* [13, 12, 15, 16, 27, 28, 29] and the work of this author [14, 25, 24, part II].

A different approach altogether is to consider **letrec** terms as *regular* terms in the sense of Courcelle [8] since these infinite terms have a finite number of distinct subterms—as we might expect from the fact that it is derived from a finite term.

Structure of the Paper

In §2 we summarize the λ-calculus as it is usually described using an implicit notion of substitution, we define acyclic and cyclic simultaneous substitutions in this tradition, and we extend standard results to these. In §3 we then discuss **let** and the relation between interpretations based on simulation through the λ-calculus and using explicit acyclic substitutions. In §4 we repeat the exercise for **letrec** interpretations where we need explicit cyclic substitutions. After concluding we discuss further work, and finally we briefly mention the relation to graph rewriting systems in an appendix.

2 Implicit Substitution and β-reduction

This section summarizes the λ-calculus and traditional λ-reduction based on "implicit" substitutions, *i.e.*, on substitution as metanotation. We first define the set Λ of λ-terms modulo renaming equipped with acyclic and cyclic simultaneous substitution. Then we define traditional β-reduction in terms of substitution and formulate usual theorems in the tradition of Barendregt [5].

2.1 Definition. Assume a set $\mathbb{V}$ of variables (ranged over by $x, y, z, \ldots$).

(i) The *λ-terms* (ranged over by $M, N, P, \ldots$) are defined inductively by

$$M \quad ::= \quad x \quad | \quad (\lambda x.M) \quad | \quad (MN)$$

We omit outermost ()s and ()s that may be placed using the following rules: (a) abstraction associates to the right, (b) application associates to the left, and (c) application takes precedence over abstraction. *E.g.*, $(\lambda xy.MNP) = (\lambda x.(\lambda y.((MN)P)))$.

(ii) The *free variables* of a term is the set of variables not bound by an enclosing abstraction; inductively:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) \setminus \{x\} \\ \mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{aligned}$$

(iii) A term with a *renaming* postfix $[x := y]$ denotes the term obtained by changing 'free occurrences' of x in it to y; inductively specified as $\Rrightarrow$ (assuming x, y, z are distinct variables) by:

$$\begin{aligned} x\,[x := y] &\Rrightarrow y \\ y\,[x := y] &\Rrightarrow y \\ z\,[x := y] &\Rrightarrow z \\ (\lambda x.M)\,[x := y] &\Rrightarrow \lambda x.M \\ (*) \qquad (\lambda y.M)\,[x := y] &\Rrightarrow \lambda y'.\,M[y := y'][x := y] \quad \text{if } y' \notin \mathrm{FV}(M) \\ (\lambda z.M)\,[x := y] &\Rrightarrow \lambda z.\,M[x := y] \\ (MN)\,[x := y] &\Rrightarrow M[x := y]\,N[x := y] \end{aligned}$$

Note: We use $\Rrightarrow$ to indicate the direction that should be used to interpret the renaming as a 'pure' λ-term.

(iv) *Equivalence* of terms means "syntactic equality modulo renaming" (*aka* "α-equivalence"), and is defined inductively by

$$\begin{aligned} x &\equiv x \\ \lambda x.M &\equiv \lambda y.N \quad \text{if } M[x := y] \equiv N \\ MN &\equiv PQ \quad\;\; \text{if } M \equiv P \wedge N \equiv Q \end{aligned}$$

Clearly $M \Rrightarrow N$ implies $M \equiv N$.

(v) Λ is the set of *opaque λ-terms*, *i.e.*, the λ-terms modulo equivalence: $\Lambda = \lambda\text{-}terms/{\equiv}$.

The definition of renaming above carefully avoids 'variable capture,' *i.e.*, changing a free variable into a bound variable, at the cost of introducing an extra renaming (equation $(*)$ of 2.1(iii) adds $[y := y']$ to rename the bound variables before it happens). In particular this means that every opaque λ-term has representative λ-terms where all bound variables are different. We will make use of this below.

2.2 Convention. All considerations in the following are over Λ, *i.e.*, we will consider λ-terms as representatives for the corresponding opaque λ-term. In particular sideconditions on freeness of variables introduced by "with" will always be assumed satisfied by picking an appropriate representative (this is essentially the *variable convention* [5, conventions 2.1.12–13]).

With this convention we are ready for a definition of 'proper' *simultaneous substitution* based on the 'variable substitution' (renaming) defined above.

2.3 Definitions.

(i) A *binder* is a pair $y := Q$ of a variable and a term. A *binder set* is an unordered collection of the form $y_1 := Q_1, y_2 := Q_2, \ldots, y_k := Q_k$ for some integer $k \geq 1$. We abbreviate binder sets as $y_1 := Q, \ldots_k$.

(ii) A term with an *acyclic simultaneous substitution* postfix $[y_1 := Q_1, \ldots_k]$ denotes the term obtained by replacing free occurrences of the variables $y_1, \ldots, y_k$ by the corresponding 'substituend' terms $Q_1, \ldots, Q_k$:

$$x\,[y_1 := Q_1, \ldots_k] \Rrightarrow x \tag{1}$$

$$y_i\,[y_1 := Q_1, \ldots_k] \Rrightarrow Q_i \tag{2}$$

$$(\lambda x.M)\,[y_1 := Q_1, \ldots_k] \Rrightarrow \lambda x'.\,M[x := x'][y_1 := Q_1, \ldots_k] \quad \text{with } x' \notin \mathrm{FV}(M\,Q_1 \ldots Q_k) \tag{3}$$

$$(MN)\,[y_1 := Q_1, \ldots_k] \Rrightarrow M[y_1 := Q_1, \ldots_k]\,N[y_1 := Q_1, \ldots_k] \tag{4}$$

(iii) A term with a *cyclic simultaneous substitution* postfix $[\![y_1 := Q_1, \ldots_k]\!]$, where the Q_i should be *proper terms*, *i.e.*, may not be any of the single variable terms $y_1, \ldots, y_k$, denotes the term obtained by replacing free occurrences of $y_1, \ldots, y_k$ in it by the corresponding 'substituend' terms Q_1,

$\dots, Q_k$ infinitely:

$$
\begin{array}{lrl}
(1) & x\,[\![y_1 := Q_1,\dots_k]\!] & \Rrightarrow x \\
(2) & y_i\,[\![y_1 := Q_1,\dots_k]\!] & \Rrightarrow Q_i[\![y_1 := Q_1,\dots_k]\!] \\
(3) & (\lambda x.M)\,[\![y_1 := Q_1,\dots_k]\!] & \Rrightarrow \lambda x'.\,M[x := x'][\![y_1 := Q_1,\dots_k]\!] \\
 & & \qquad \text{with } x' \notin \mathrm{FV}(M\,Q_1\dots Q_k) \\
(4) & (M N)\,[\![y_1 := Q_1,\dots_k]\!] & \Rrightarrow M[\![y_1 := Q_1,\dots_k]\!]\,N[\![y_1 := Q_1,\dots_k]\!]
\end{array}
$$

(iv) Given a term by a cyclic simultaneous substitution $M[\![y_1 := Q_1,\dots_k]\!]$. y_{i_0} is a *cyclic variable* in a cycle of length n if there is a sequence $y_{i_0} \in \mathrm{FV}(Q_{i_1})$, $y_{i_1} \in \mathrm{FV}(Q_{i_2})$, ..., $y_{i_n} \in \mathrm{FV}(Q_{i_0})$ and $\{y_{i_0},\dots,y_{i_n}\} \cap \mathrm{FV}(M) \neq \emptyset$.

Again we avoid variable capture carefully by the somewhat 'brute force' approach of renaming the bound variable of *any* abstraction under which we substitute—it is clear that this does not change the opaque λ-term in question. This problem would dissappear altogether if we used de Bruijn indices [7], but we prefer proper variables to keep the connection to **let** and **letrec** obvious.

We have used $\Rrightarrow$ again to indicate the *direction* of the substitution unfolding here as well. The following properties of the defined renaming and substitution concepts are easy to verify and will prove useful later. They are generalisations of standard λ-calculus results to simultaneous acyclic and cyclic substitutions; the only complication is the technical restriction on the substituends of cyclic substitutions that is needed to ensure that all substitutions can be unfolded. As an example, $x[\![x := x]\!]$ is not a term in this calculus (so there is no need for special "$\circ$-terms" as found in [4]).

2.4 Propositions.

(i) *The unfolding sequence* $M[y_1 := Q_1,\dots_k] \Rrightarrow \dots \Rrightarrow N$ *is always finite.*

(ii) *The unfolding sequence* $M[\![y_1 := Q_1,\dots_k]\!] \Rrightarrow \dots \Rrightarrow N$ *is finite iff there are no cyclic variables in the substitution.*

(iii) *Acyclic substitution lemma: If* $x_1,\dots,x_k \notin \mathrm{FV}(Q_1\dots Q_k)$ *then*

$$
\begin{aligned}
&M[x_1 := P_1,\dots_k][y_1 := Q_1,\dots_n] \\
&\qquad\qquad \equiv M[y_1 := Q_1,\dots_n][x_1 := P_1[y_1 := Q_1,\dots_n],\dots_k]
\end{aligned}
$$

(iv) *Cyclic unfolding lemma:*

$$M[\![x_1 := P_1,\dots_k]\!] \equiv M[x_1 := P_1,\dots_k][\![x_1 := P_1,\dots_k]\!]$$

(v) *Cyclic substitution lemma:: If* $x_1,\dots,x_k \notin \mathrm{FV}(Q_1\dots Q_k)$ *then*

$$
\begin{aligned}
&M[\![x_1 := P_1,\dots_k]\!][\![y_1 := Q_1,\dots_n]\!] \\
&\qquad\qquad \equiv M[\![y_1 := Q_1,\dots_n]\!][\![x_1 := P_1[\![y_1 := Q_1,\dots_n]\!],\dots_k]\!]
\end{aligned}
$$

The formulation of the Church-Rosser theorem for β-reduction looks as follows in our notation.

2.5 Definition. The semantics of the λ-calculus is given by the *β-reduction step* relation

$$(\beta) \qquad (\lambda x.M)N \Rightarrow M[x := N]$$

We will also use the following derived relations: *reduction* $\twoheadrightarrow$ is $\Rightarrow$s transitive, reflexive closure, and *conversion* $\twoheadleftarrow\!\!\twoheadrightarrow$ is $\Rightarrow$s transitive, reflexive, and symmetric closure. M is a *redex* iff $\exists N\colon M \Rightarrow N$, otherwise it is a *normal form*: $\mathrm{nf}(M)$. Finally, *completion* $\twoheadrightarrow\!\!|$ is defined by $M \twoheadrightarrow\!\!| \; N$ iff $M \twoheadrightarrow N$ and $\mathrm{nf}(N)$.

Remark. We silently impose the usual restriction that a "notion of reduction" relation must always be transitive, reflexive, and *compatible* with (the operations of) the set it is defined over—the last simply means that the reduction step can be applied anywhere in terms. Other notions are possible, *cf.* Abramsky [2].

2.6 Theorem (Church-Rosser). *β-reduction is confluent, i.e., completes the 'diamond diagram'*

$$\begin{tikzcd}
 & M \arrow[dl, Rightarrow, two heads] \arrow[dr, Rightarrow, two heads] & \\
N_1 \arrow[dr, Rightarrow, two heads, dashed] & & N_2 \arrow[dl, Rightarrow, two heads, dashed] \\
 & R &
\end{tikzcd}$$

It is trivial to show that $Q_1 \Rightarrow Q_1'$ implies $M[y_1 := Q_1, \ldots_k] \twoheadrightarrow M[y_1 := Q_1', \ldots_k]$, and the following are easy to verify:

2.7 Propositions. *β-reduction is substitutive with respect to both acyclic and cyclic substitution:*

(i) $M \twoheadrightarrow N$ *implies* $M[y_1 := Q_1, \ldots_k] \twoheadrightarrow N[y_1 := Q_1, \ldots_k]$.
(ii) $M \twoheadrightarrow N$ *implies* $M\llbracket y_1 := Q_1, \ldots_k\rrbracket \twoheadrightarrow N\llbracket y_1 := Q_1, \ldots_k\rrbracket$.

However, $Q_1 \twoheadrightarrow Q_1'$ does not in general imply $M\llbracket y_1 := Q_1, \ldots_k\rrbracket \twoheadrightarrow M\llbracket y_1 := Q_1', \ldots_k\rrbracket$ because Q_1' may be a variable making the second substitution undefined. *E.g.*, even though $Ix \twoheadrightarrow x$ we do not have that $x\llbracket x := Ix\rrbracket \equiv I(I(I\ldots))$ (where $I \equiv \lambda y.y$) reduces to $x\llbracket x := x\rrbracket$ since the latter is not a term.

3 Explicit Acyclic Substitution and "let"

We will now concentrate on the interpretation of **let** and acyclic substitution. First we show how we may simulate simultaneous substitution using β-reduction. Then we show how the mechanics of substitution may be build in by presenting an alternate formulation of the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien, and Lévy [1].

Intuitively the denotation of **let** should be a standard λ-term with the same meaning as an acyclic substitution, *i.e.*,

$$\textbf{let } x_1 = N_1 \textbf{ and} \ldots \textbf{and } x_k = N_k \textbf{ in } M \;\equiv\; M[x_1 := N_1, \ldots_k]$$

It is easy to prove that such a term exist using the standard 'tupling trick.'

3.1 Definition. *k-tupling* $\langle M_1, \ldots, M_k\rangle$ of $M_1, \ldots, M_k$ with *projections* π^k_n, $1 \le n \le k$, are terms that satisfy $\pi^k_n \langle M_1, \ldots, M_k\rangle \twoheadrightarrow M_n$.

In fact simple versions of tupling exist, *e.g.*, $\langle M_1, \ldots, M_k\rangle \equiv \lambda z.zM_1 \ldots M_k$ with $\pi^k_n \equiv \lambda x.x(\lambda z_1 \ldots z_k.z_n)$.

3.2 Lemma. *k-tupling simulates acyclic simultaneous substitution:*

$$(\lambda y.P[y_1 := y\,\pi^k_1, \ldots, y_k := y\,\pi^k_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \mathrel{\twoheadrightarrow\!\!|} P[y_1 := Q_1, \ldots_k]$$

Proof. Proposition 2.4(i) ensures that it is sufficient to complete

$$\begin{tikzcd}
P[y_1 := Q_1, \ldots_k] \arrow[r, Rightarrow] \arrow[d, dash, "\vdash"] & \Box \arrow[r, Rightarrow] \arrow[d, dash, "\vdash"] & \Box \arrow[r, Rightarrow] \arrow[d, dash, "\vdash"] & \cdots \arrow[r, Rightarrow] & P' \arrow[d, phantom, "\equiv"] \\
R_0 \arrow[r, twoheadrightarrow] & R_1 \arrow[r, twoheadrightarrow] & R_2 \arrow[r, twoheadrightarrow] & \cdots \arrow[r, twoheadrightarrow] & M'
\end{tikzcd}$$

where $R_0 \equiv (\lambda y.P[y_1 := y\,\pi^k_1, \ldots, y_k := y\,\pi^k_k])\ (\lambda p.p\langle Q_1, \ldots, Q_k\rangle)$ and $P[y_1 := Q_1, \ldots_k] \vdash R$ means that R represents that particular 'unfoldedness' of the substitution. We are finished when we have proven the correctness of all possible such diagram 'cells' by induction over the structure of definition 2.3(ii): for each possible $\Rightarrow$-step we prove that there is a corresponding $\twoheadrightarrow$-reduction from the representation 'before' to the representation 'after.'

Case 1: $x[y_1 := Q_1, \ldots_k] \vdash (\lambda y.x[y_1 := y\,\pi^k_1, \ldots_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \Rightarrow x$ ✓

Case 2:

$$\begin{aligned}
&y_i[y_1 := Q_1, \ldots_k] \\
&\quad \vdash (\lambda y.y_i[y_1 := y\,\pi^k_1, \ldots_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \\
&\quad \Rightarrow y_i[y_1 := y\,\pi^k_1, \ldots_k][y := \lambda p.p\langle Q_1, \ldots, Q_k\rangle] \\
&\quad \Rightarrow (y\,\pi^k_i)[y := \lambda p.p\langle Q_1, \ldots, Q_k\rangle] \\
&\quad \Rightarrow (\lambda p.p\langle Q_1, \ldots, Q_k\rangle)\,\pi^k_i \\
&\quad \Rightarrow \pi^k_i\langle Q_1, \ldots, Q_k\rangle \twoheadrightarrow Q_i \quad ✓
\end{aligned}$$

Case 3:

$$\begin{aligned}
&(\lambda x.M)[y_1 := Q_1, \ldots_k] \\
&\quad \vdash (\lambda y.(\lambda x.M))[y_1 := y\,\pi^k_1, \ldots_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \\
&\quad \Rightarrow (\lambda y.\lambda x'.M[x := x'][y_1 := y\,\pi^k_1, \ldots_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \\
&\quad \Rightarrow \lambda x'.M[x := x'][y_1 := (\lambda p.p\langle Q_1, \ldots, Q_k\rangle)\,\pi^k_1, \ldots_k] \\
&\quad \Rightarrow \lambda x'.M[x := x'][y_1 := \pi^k_1\langle Q_1, \ldots, Q_k\rangle, \ldots_k] && \text{by 2.4(iii)} \\
&\quad \twoheadrightarrow \lambda x'.M[x := x'][y_1 := Q_1, \ldots_k] && \text{by 2.4(iii)} \quad ✓
\end{aligned}$$

Case 4:

$$\begin{aligned}
&(MN)[y_1 := Q_1, \ldots_k] \\
&\quad \vdash (\lambda y.(MN)[y_1 := y\,\pi^k_1, \ldots_k])(\lambda p.p\langle Q_1, \ldots, Q_k\rangle) \\
&\quad \Rightarrow (MN)[y_1 := y\,\pi^k_1, \ldots_k][y := (\lambda p.p\langle Q_1, \ldots, Q_k\rangle)] \\
&\quad \Rightarrow (MN)[y_1 := (\lambda p.p\langle Q_1, \ldots, Q_k\rangle)\,\pi^k_1, \ldots_k] \\
&\quad \twoheadrightarrow (MN)[y_1 := Q_1, \ldots_k] && \text{by 2.4(iii)} \\
&\quad \Rightarrow M[y_1 := Q_1, \ldots_k]\,N[y_1 := Q_1, \ldots_k] \quad \blacksquare
\end{aligned}$$

This shows that the above encoding of **let** is correct and that the 'interpretation overhead' using tupling is constant (actually linear in the number k of simultaneous binders). It also shows that the expressive power of the standard λ-calculus will not be extended when we add substitution explicitly.

3.3 Definition. Assume definition 2.1. The $\lambda\sigma$-calculus is derived by modifying the individual points as follows:

(i) The *$\lambda\sigma$-terms*: add acyclic binding as syntax, *i.e.*,

$$M \;::=\; x \;\mid\; (\lambda x.M) \;\mid\; (MN) \;\mid\; M[x_1 := N_1, \dots_k]$$

(ii) The *free variables*: add

$$\mathrm{FV}(M[x_1 := N_1, \dots_k]) = (\mathrm{FV}(M) \setminus \{x_1, \dots, x_k\}) \cup \mathrm{FV}(N_1 \dots N_k)$$

(iii) The *renaming* postfix: add

$$\begin{aligned}
&M[x_1 := N_1, \dots_k]\,[x_i := y] \;\Rrightarrow\; M[x_1 := N_1, \dots_k]\\
&M[y_1 := N_1, \dots_k]\,[x := y_i]\\
&\qquad \Rrightarrow M[y_1 := y_1'] \dots [y_k := y_k'][x := y_i][y_1' := N_1, \dots_k]\\
&M[z_1 := N_1, \dots_k]\,[x := y] \quad \Rrightarrow M[x := y][z_1 := N_1, \dots_k]
\end{aligned}$$

(iv) *Equivalence*: add

$$M[x_1 := N_1, \dots_k] \equiv P[y_1 := Q_1, \dots_k] \quad \text{if } M[x_1 := y_1] \dots [x_k := y_k] \equiv P \wedge \forall i\colon N_i \equiv Q_i$$

The *$\lambda\sigma$-reduction steps* are β-reduction (of definition 2.5) and syntactic versions of acyclic $\Rrightarrow$-unfolding (of definition 2.3(ii)):

$(\beta\sigma)$ $\qquad (\lambda x.M)N \Rightarrow M[x := N]$

(σ_0) $\qquad x[y_1 := Q_1, \dots_k] \Rightarrow x$

(σ_1) $\qquad y_i[y_1 := Q_1, \dots_k] \Rightarrow Q_i$

(σ_2) $\qquad (\lambda x.M)[y_1 := Q_1, \dots_k] \Rightarrow \lambda x'.\,M[x := x'][y_1 := Q_1, \dots_k]$ with $x' \notin \mathrm{FV}(M\,Q_1 \dots Q_k)$

(σ_3) $\qquad (MN)[y_1 := Q_1, \dots_k] \Rightarrow M[y_1 := Q_1, \dots_k]\,N[y_1 := Q_1, \dots_k]$

The theory $\boldsymbol{\sigma}$ contains the σ-rules and $\boldsymbol{\lambda\sigma}$ contains all the above rules.

The remainder of this section is devoted to prove the confluence of $\boldsymbol{\lambda\sigma}$: first we prove that $\boldsymbol{\sigma}$ and $(\beta\sigma)$ seperately are confluent and strongly normalizing. Then we prove that they commute and consequently that $\boldsymbol{\lambda\sigma}$ is confluent. It is interesting to compare this proof with the confluence of the untyped $\lambda\sigma$ of [1] because their proof combines the *full* β-reduction with σ-reduction whereas our proof combines the *syntactic* $(\beta\sigma)$ with the (also syntactic) substitution rules $\boldsymbol{\sigma}$.

The key lemma making this possible is Hindley–Rosen [5, proposition 3.3.5] presented immediately hereafter. Notice how our presentation below is rather terse because most of the properties we prove are easily seen to be equivalent to standard substitution results above! This is the obvious advantage of the fact that $\lambda\sigma$ in a way is just a syntactic version of the standard calculus.

3.4 Lemma (Hindley–Rosen). *Given two reduction relations* $\twoheadrightarrow_1$ *and* $\twoheadrightarrow_2$ *and let* $\twoheadrightarrow_{12}$ *be the transitive reflexive closure of their union. Suppose* (1) *that each of the two relations are confluent and* (2) *that they commute, i.e.,*

$$(1)\ \begin{tikzcd} & M \arrow[dl, two heads, "1"'] \arrow[dr, two heads, "1"] & \\ M_1 \arrow[dr, two heads, "1"'] & & M_2 \arrow[dl, two heads, "1"] \\ & N & \end{tikzcd} \wedge \begin{tikzcd} & M \arrow[dl, two heads, "2"'] \arrow[dr, two heads, "2"] & \\ M_1 \arrow[dr, two heads, "2"'] & & M_2 \arrow[dl, two heads, "2"] \\ & N & \end{tikzcd}, \text{ and } (2)\ \begin{tikzcd} & M \arrow[dl, two heads, "1"'] \arrow[dr, two heads, "2"] & \\ M_1 \arrow[dr, two heads, "2"'] & & M_2 \arrow[dl, two heads, "1"] \\ & N & \end{tikzcd},$$

then their union is confluent:

$$\begin{tikzcd} & M \arrow[dl, two heads, "12"'] \arrow[dr, two heads, "12"] & \\ M_1 \arrow[dr, two heads, "12"'] & & M_2 \arrow[dl, two heads, "12"] \\ & N & \end{tikzcd}.$$

3.5 Theorem. $\boldsymbol{\lambda\sigma}$ *is confluent.*

Proof. Use Hindley–Rosen 3.4 on $(\beta\sigma)$ and $\boldsymbol{\sigma}$: We have

$$\begin{tikzcd} & M \arrow[dl, Rightarrow, two heads, "\beta\sigma"'] \arrow[dr, Rightarrow, two heads, "\beta\sigma"] & \\ M_1 \arrow[dr, Rightarrow, two heads, "\beta\sigma"'] & & M_2 \arrow[dl, Rightarrow, two heads, "\beta\sigma"] \\ & N & \end{tikzcd}, \begin{tikzcd} & M \arrow[dl, Rightarrow, two heads, "\sigma"'] \arrow[dr, Rightarrow, two heads, "\sigma"] & \\ M_1 \arrow[dr, Rightarrow, two heads, "\sigma"'] & & M_2 \arrow[dl, Rightarrow, two heads, "\sigma"] \\ & N & \end{tikzcd}, \text{and } \begin{tikzcd} & M \arrow[dl, Rightarrow, two heads, "\beta\sigma"'] \arrow[dr, Rightarrow, two heads, "\sigma"] & \\ M_1 \arrow[dr, Rightarrow, two heads, "\sigma"'] & & M_2 \arrow[dl, Rightarrow, two heads, "\beta\sigma"] \\ & N & \end{tikzcd}$$

The left follows from the observation that $(\beta\sigma)$-reduction will replace some β-redex $(\lambda x.M)N$ with σ-redex $M[x := N]$; whether M and N contain β-redices themself is of no consequence. Since there are no more than a finite number of redexes in any term then this is also strongly normalizing. That $\boldsymbol{\sigma}$ is strongly normalizing follows immediately from proposition 2.4(i); the confluence then follows from a simple induction over the definition (it also follows from 2.6 and 3.2). Finally the leftmost diagram is just a rephrasing of 2.4(iii). ■

Thus it seems that standard concepts of the λ-calculus extend nicely to handle acyclic substitutions allowing the modeling of sharing through **let**.

4 Explicit Cyclic Substitution and "letrec"

In this section we will discuss the **letrec** interpretation, prove its correctness as far as it goes, and discuss the problems with it. Finally we present the $\lambda\sigma^*$ calculus, a cyclic version of $\lambda\sigma$ defined in the previous section, and prove that it is confluent.

Intuitively we should encode **letrec** like **let** but with a simultaneous cyclic substitution, *i.e.*,

$$\textbf{letrec } x_1 = N_1 \textbf{ and} \ldots \textbf{and } x_k = N_k \textbf{ in } M \ \equiv\ M[\![x_1 := N_1, \ldots_k]\!]$$

The problem with this is, of course that *any* term created by a cyclic substitution containing cylic variables is infinite as the *parity* example of the introduction showed. In fact it is clear that the definition of **letrec** terms may yield infinite terms in much the same way as fixed point induction. The following example shows how this may be exploited to encode an 'immediate fixed point' operator, assuming $x \notin \mathrm{FV}(M)$:

$$\begin{aligned}
\textbf{letrec } x = Mx \textbf{ in } x &\equiv x[\![x := Mx]\!] \\
&\Rrightarrow Mx[\![x := Mx]\!] \\
&\Rrightarrow M(Mx)[\![x := Mx]\!] \\
&\;\vdots \\
&\Rrightarrow M(M(M(M\ldots)))[\![x := Mx]\!] \\
&\Rrightarrow M(M(M(M\ldots)))
\end{aligned}$$

The above involves an infinite number of $\Rrightarrow$ unfoldings to build the infinite term, of course; the last step is due to the fact that there are no free xs in the infinite sequence of Ms. Thus our simulation result is weaker since we can not hope to simulate infinite unfolding.

4.1 Lemma. *β-reduction using fixed point iteration and k-tupling may simulate the termination or nontermination of cyclic simultaneous substitution as follows depending on presense of cyclic variables in $P[\![y_1 := Q_1, \ldots_k]\!]$:*

$$\begin{tikzcd}
P[\![y_1 := Q_1, \ldots_k]\!] \arrow[r, Rrightarrow] & \square \arrow[r, Rrightarrow] & \square \arrow[r, Rrightarrow] & \cdots \arrow[r, Rrightarrow] & P' \arrow[d, phantom, "\equiv" rotate=90] & \text{iff not cyclic} \\
R_0 \arrow[u, -|] \arrow[r, Rightarrow, two heads] & R_1 \arrow[u, dashed, -|] \arrow[r, Rightarrow, two heads] & R_2 \arrow[u, dashed, -|] \arrow[r, Rightarrow, two heads] & \cdots \arrow[r, Rightarrow, two heads] & R' &
\end{tikzcd}$$

where $R_0 = \overline{P}[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)]$, $\forall M\colon \overline{M} \equiv M[y_1 := y\,\pi^k_1, \ldots, y_k := y\,\pi^k_k]$.

Proof. Proceeds along the same lines as the proof for 3.2 except that we use proposition 2.4(v) which means that the second case is the only one that is substantially different—this is as should be expected since it corresponds to the only equation of 2.3(iii) that differs from 2.3(ii):

Case 2:

$$\begin{aligned}
&y_i[\![y_1 := Q_1, \ldots_k]\!] \\
&\qquad\vdash \overline{y_i}[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)] \\
&\qquad\Rightarrow (y\,\pi^k_i)[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)] \\
&\qquad\Rightarrow Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)\,\pi^k_i \\
&\qquad\twoheadrightarrow (\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)(Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle))\,\pi^k_i \\
&\qquad\Rightarrow (\lambda p.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)]\,\pi^k_i \\
&\qquad\twoheadrightarrow (\lambda p.(p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)])\,\pi^k_i \\
&\qquad\Rightarrow (p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)[y := Y(\lambda yp.p\langle\overline{Q_1}, \ldots, \overline{Q_k}\rangle)][p := \pi^k_i]
\end{aligned}$$

$$
\begin{aligned}
&\Rightarrow (p[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)]\,\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle \\
&\qquad\quad [y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)])[p := \pi_i^k] \\
&\Rightarrow (p\,\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)])[p := \pi_i^k] \\
&\Rightarrow (p[p := \pi_i^k]\,\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)][p := \pi_i^k]) \\
&\twoheadRightarrow (p[p := \pi_i^k]\,\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)]) \\
&\Rightarrow (\pi_i^k\,\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)]) \\
&\twoheadRightarrow \pi_i^k\,\langle\overline{Q_1}[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)],\ldots, \\
&\qquad\quad \overline{Q_k}[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)]\rangle \\
&\twoheadRightarrow \overline{Q_i}[y := Y(\lambda yp.p\langle\overline{Q_1},\ldots,\overline{Q_k}\rangle)] \quad \checkmark
\end{aligned}
$$

This reduction sequence also shows how all cyclic variables remain in the term after unfolding. ■

Now let us see what happens if we add cyclic substitution as *syntax* to the calculus.

4.2 Definition. Assume definition 2.1. The $\lambda\sigma^*$-*calculus* is derived by modifying the individual points as follows:

(i) The $\lambda\sigma$-*terms*: add cyclic binding as syntax, *i.e.*,

$$M \quad ::= \quad x \quad | \quad (\lambda x.M) \quad | \quad (MN) \quad | \quad M[\![x_1 := N_1,\ldots{}_k]\!]$$

we say that a term is *cyclic* when it contains syntax that would be cyclic as a substitution.

(ii) The *free variables*: add

$$\mathrm{FV}(M[\![x_1 := N_1,\ldots{}_k]\!]) = \mathrm{FV}(M\,N_1\ldots N_k) \setminus \{x_1,\ldots,x_k\}$$

(iii) The *renaming* postfix: add

$$
\begin{aligned}
&M[\![x_1 := N_1,\ldots{}_k]\!]\,[x_i := y] \;\twoheadRightarrow M[\![x_1 := N_1,\ldots{}_k]\!] \\
&M[\![y_1 := N_1,\ldots{}_k]\!]\,[x := y_i] \\
&\qquad \twoheadRightarrow M[y_1 := y_1']\ldots[y_k := y_k'][x := y_i][\![y_1' := N_1,\ldots{}_k]\!] \\
&M[\![z_1 := N_1,\ldots{}_k]\!]\,[x := y] \quad \twoheadRightarrow M[x := y][\![z_1 := N_1,\ldots{}_k]\!]
\end{aligned}
$$

(iv) *Equivalence*: add

$$
\begin{aligned}
&M[\![x_1 := N_1,\ldots{}_k]\!] \equiv P[\![y_1 := Q_1,\ldots{}_k]\!] \\
&\qquad \text{if } M[x_1 := y_1]\ldots[x_k := y_k] \equiv P \wedge \forall i\colon N_i[x_i := y_i]\ldots[x_k := y_k] \equiv Q_i
\end{aligned}
$$

The $\lambda\sigma^*$-*reduction steps* are β-reduction (of definition 2.5) and syntactic versions of cyclic $\Rrightarrow$-unfolding (of definition 2.3(iii)):

$$
\begin{array}{lrl}
(\beta\sigma^*) & (\lambda x.M)N & \Rightarrow M[x := x'][\![x' := N]\!] \\
 & & \quad \text{with } x' \notin \mathrm{FV}(MN) \\
(\sigma_0^*) & x[\![y_1 := Q_1, \ldots_k]\!] & \Rightarrow x \\
(\sigma_1^*) & y_i[\![y_1 := Q_1, \ldots_k]\!] & \Rightarrow Q_i[\![y_1 := Q_1, \ldots_k]\!] \\
(\sigma_2^*) & (\lambda x.M)[\![y_1 := Q_1, \ldots_k]\!] & \Rightarrow \lambda x'.M[x := x'][\![y_1 := Q_1, \ldots_k]\!] \\
 & & \quad \text{with } x' \notin \mathrm{FV}(M\,Q_1 \ldots Q_k) \\
(\sigma_3^*) & (MN)[\![y_1 := Q_1, \ldots_k]\!] & \Rightarrow M[\![y_1 := Q_1, \ldots_k]\!]N[\![y_1 := Q_1, \ldots_k]\!]
\end{array}
$$

The theory $\boldsymbol{\sigma}^*$ contains the σ^*-rules and $\boldsymbol{\lambda\sigma}^*$ contains all the above rules.

The Church-Rosser theorem for $\boldsymbol{\lambda\sigma}^*$ is similar to theorem 3.5:

4.3 Theorem. $\boldsymbol{\lambda\sigma}^*$ *is confluent.*

Proof sketch. We invoke the Hindley–Rosen lemma 3.4 again, this time with $(\beta\sigma^*)$ and $\boldsymbol{\sigma}^*$. Again $(\beta\sigma^*)$ is confluent and strongly normalizing, and from 2.4(iv), 3.5, and 4.1 we conclude that $\boldsymbol{\sigma}^*$ is confluent even though it is strongly normalizing exactly for non-cyclic terms and divergent for cyclic terms. □

So it also seems that standard concepts of the λ-calculus extend nicely to cyclic substitutions allowing the modeling of recursive definitions through **letrec**.

5 Conclusion and Future Work

We have discussed different ways of describing sharing and recursion in the λ-calculus, and have shown proofs of confluency. We would like to improve the understanding as well as the presentation; in particular it appears promising to investigate the relation between fixed point iteration and the infinite unfolding we have presented. The knowledge gained is that *explicitly represented sharing and recursion* can be maintained and exploited in a computational model. This will aid in the creation of formal links between the 'operational insight' of implementors and the 'denotational oversight' of formalists.

The natural next step is thus to try to 'derive' an abstract reduction machine from our very simple $\lambda\sigma$ and $\lambda\sigma^*$-rewrite systems in a way similar to the way the Krivine machine is derived by Curien [9] from his $\lambda\rho$-calculus. It will be interesting to see how cyclic structures show up in such a machine. It would also be interesting to try to combine current work on optimal reductions in acyclic explicit substitution calculi with the cyclicity notions discussed here. Finally the relation between this presentation, regular trees, and in particular graph reduction needs to be better understood—it unfortunately seems that there is a lot of duplication of work due to the lack of a current reference frame.

Appendix. Graph Reduction

We mentioned in the introduction that our techniques are similar to operational techniques used within "graph reduction machine" descriptions of functional programming languages. A different approach is to describe sharing by using a *term graph* representation. The author's thesis [24, part II] shows how the substitution concepts above may be 'lifted' to term graph rewriting [6] such that we may represent the β-reduction rule as

$$(\boldsymbol{\beta})\qquad \begin{tikzcd} & & @ \arrow[dl] \arrow[dr] & & \\ & \lambda \arrow[dl] \arrow[dr] & & \boldsymbol{a} & \Rightarrow \quad \boldsymbol{b}[\boldsymbol{x} := \boldsymbol{a}] \\ \boldsymbol{x} & & \boldsymbol{b} & & \end{tikzcd}$$

in a suitable representation (where λ denotes abstraction, @ denotes application, bold variables denote subgraphs, and the substitution notation $\_[\_ := \_]$ is suitably defined over graphs). The difficult bit is of course to define the substitution such that it faithfully generalizes standard substitution to the much richer domain of graphs. And the interesting but perhaps not suprising thing about generalizing term to graph rewriting is that the encountered difficulties resemble the restriction of definition 2.3(iii) that substituends of a simultaneous cyclic substitution postfix can not be the substitution variables, *cf.* [15].

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, *Explicit Substitutions*, in [22], 31–46.
2. S. Abramsky, *The Lazy Lambda Calculus*, ch. 4, in D. A. Turner (ed.), *Research Topics in Functional Programming*, Addison-Wesley, 1990, pp. 65–116.
3. Z. M. Ariola and Arvind, *A Syntactic Approach to Program Transformations*, in *PEPM '91—Symposium on Partial Evaluation and Semantics-based Program Manipulation* (Yale University, New Haven, Connecticut, USA), 17–19 June 1991, pp. 116–129.
4. ———, *Graph Rewriting Systems*, Tech. Report, MIT, 1992, to appear.
5. H. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, revised edition, North-Holland, 1984.
6. H. P. Barendregt, M. C. D. J. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep, *Term Graph Rewriting*, in J. W. de Bakker, A. J. Nijman, and P. C. Treleaven (eds.), *PARLE '87—Parallel Architectures and Languages Europe vol. II* (Eindhoven, The Netherlands), LNCS no. 256, Springer-Verlag, June 1987, pp. 141–158.
7. N. G. de Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem*, Koninkijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences **75** (1972), 381–392.
8. B. Courcelle, *Fundamental Properties of Infinite Trees*, Theoretical Computer Science **25** (1983), 95–169.

9. P.-L. Curien, *An Abstract Framework for Environment Machines*, Unpublished note from LIENS/CNRS, July 1990.
10. G. Gonthier, M. Abadi, and J.-J. Lévy, *The Geometry of Optimal Lambda Reduction*, in *POPL '92—Nineteenth Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico), January 1992, pp. 1–14.
11. K. Grue, *Call-by-Mix: A Reduction Strategy for Pure λ-calculus*, Unpublished note from DIKU (University of Copenhagen), 1987.
12. C. Hankin, *Static Analysis of Term Graph Rewriting Systems*, ESPRIT "Semantique" working paper, 1990.
13. C. A. R. Hoare, *Recursive Data Structures*, Journal of Computer and Information Sciences **4** (1975), no. 2, 105–132.
14. K. H. Holm, *Graph Matching in Operational Semantics and Typing*, in A. Arnold (ed.), *CAAP '90—15th Colloqvium on Trees and Algebra in Programming* (Copenhagen, Denmark), LNCS no. 431, Springer-Verlag, March 1990, pp. 191–205.
15. J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries, *On the Sdequacy of Graph Rewritng for Simulating Term Rewriting*, in [26] ch. 8.
16. P. W. M. Koopman, S. E. W. Smetsers, M. C. D. J. van Eekelen, and M. J. Plasmeijer, *Efficient Graph Rewriting using the Annotated Functional Strategy*, in [21], 225–250, (available as nijmegen tech. report 91-25).
17. J. Lamping, *An Algorithm for Optimal Lambda Calculus Reduction*, in [22], 16–30.
18. J.-J. Lévy, *Optimal Reductions in the Lambda Calculus*, in J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980, pp. 159–191.
19. L. Maranget, *Optimal Derivations in Weak Lambda-Calculi and in Orthogonal Terms Rewriting Systems*, in *POPL '91—Eightteenth Annual ACM Symposium on Principles of Programming Languages* (Orlando, Florida), January 1991, pp. 255–269.
20. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
21. M. J. Plasmeijer and M. R. Sleep (eds.), *SemaGraph '91 Symposium on the Semantics and Pragmatics of Generalized Graph Rewriting* (Nijmegen, Holland), December 1991, (available as nijmegen tech. report 91-25).
22. POPL, *POPL '90—Seventeenth Annual ACM Symposium on Principles of Programming Languages* (San Francisco, California), January 1990.
23. K. H. Rose, *Explicit Cyclic Substitutions*, in M. Rusinowitch and J.-L. Rémy (eds.), *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems* (Pont-a-Mousson, France), LNCS, Springer-Verlag, July 1992, also available as DIKU semantics note D–143.
24. ———, *GOS—Graph Operational Semantics*, Speciale 92-1-9, DIKU (University of Copenhagen), Universitetsparken 1, DK–2100 København Ø, Denmark, March 1992, (56pp).
25. ———, *Graph-based Operational Semantics of a Lazy Functional Language*, in [26] ch. 22, 234–247, also available as DIKU semantics note D–146.
26. M. R. Sleep, M. J. Plasmeijer, and M. C. D. J. van Eekelen (eds.), *Term Graph Rewriting: Theory and Practice*, John Wiley & Sons, 1993.
27. J. Staples, *A Graph-like Lambda Calculus for which Leftmost Outermost Reduction is Optimal*, in V. Claus, H. Ehrig, and G. Rozenberg (eds.), *1978 International Workshop in Graph Grammars and their Application to Computer Science and Biology* (Bad Honnef, F. R. Germany), LNCS no. 73, Springer-Verlag, 1978, pp. 440–454.

28. Y. Toyama, S. Smetsers, M. van Eekelen, and M. J. Plasmeijer, *The Functional Strategy and Transitive Term Rewriting Systems*, in [21], 99–114, (available as nijmegen tech. report 91-25).
29. C. P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, Ph.D. Thesis, Programming Research Group, Oxford University, 1971.