Rewriting

I. Bethke, J.W. Klop

1. Introduction

Many computations can be modeled as step-by-step transformations or rewriting of a string of symbols (words, expressions, terms), intending to reach some final result as an answer (a *normal form*). Such a rewrite step, to be perceived as an atomic computation step, consists of replacing part of the expression by a simpler part, according to the rules of some *rewriting system*. E.g., in arithmetic: $(3+5)\cdot(1+2) \to 8\cdot(1+2) \to 8\cdot 3 \to 24$.

The study of rewriting systems belongs to the area of symbolic computation. The main applications of rewriting are in the fields of abstract data types and algebraic specifications, automated theorem proving, functional programming and logic programming. Rewriting can be studied at several levels. In this informal survey, we aim to give an impression of several of these levels, roughly in order of increasing complexity. In our choice of topics, we have put an emphasis on subjects that CWI has contributed to during the last decade. Especially we mention conditional rewriting, higher-order rewriting, infinitary rewriting and term graph rewriting.

2. Abstract rewriting

The simplest level is *abstract rewriting* which consists essentially of the study of one or more binary relations on some set of abstract objects. Figure 1 displays such an abstract reduction system or ARS. The arrows give the binary rewrite relation on the four objects a, b, c, d. Thus we have *successful*

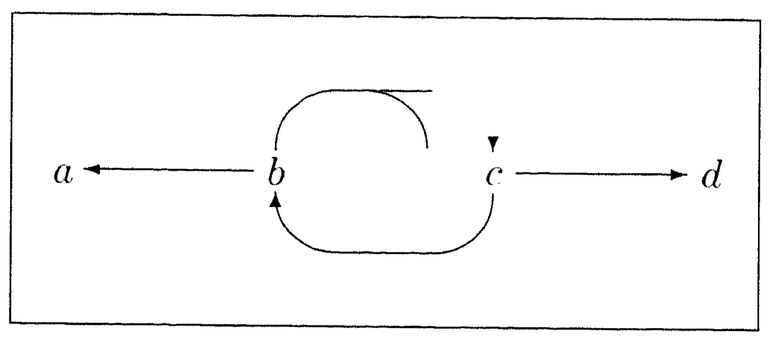


Figure 1.

terminating rewritings such as $b \to c \to d$, but also *unsuccessful* infinite rewritings $b \to c \to b \to c \to \cdots$. The elements a and d, from which no further step is possible, are called *normal forms*.

3. STRING REWRITING

A more concrete form of rewriting is that of *string rewriting*. As an example, consider the following interesting puzzle, posed by H. Zantema (Utrecht University): Given is the string rewrite rule

$$0011 \to 111000.$$

An application of the rule consists in replacing in some 0, 1-string an occurrence of a substring 0011 by 111000. For example, we may rewrite

$$\begin{aligned} &\underline{0011}1111 \to \\ &1110\underline{0011}11 \to \\ &111011\underline{0011} \to \\ &11101110111000 \end{aligned}$$

from where no further rewriting is possible; so the string is a normal form. The reader may enjoy herself with discovering that any rewrite sequence using this rule must terminate eventually - that is, the rule has the *termination property*. The proof is non-trivial.

4. FIRST-ORDER TERM REWRITING

The next level of rewriting, next in order of increasing complexity, is that of *first-order term rewriting*. Whereas strings (or words) over some alphabet are rather poorly structured carriers of information, first-order terms are a very general medium for carrying information, and this notion together with its semantics as given by A. Tarski has turned out to be extremely fruitful and permeates much of mathematical logic and computer science of this century. We introduce the notion of a first-order rewrite system by the example in Table 1.

These four rewrite rules specify elegantly addition `A` and multiplication `M` on natural numbers $0, \mathtt{S}(0), \mathtt{S}(\mathtt{S}(0)), \ldots$. Using these rules we compute

r_1	$\mathtt{A}(x, 0)$	$\rightarrow$	x
r_2	$\mathtt{A}(x, \mathtt{S}(y))$	$\rightarrow$	$\mathtt{S}(\mathtt{A}(x, y))$
r_3	$\mathtt{M}(x, 0)$	$\rightarrow$	0
r_4	$\mathtt{M}(x, \mathtt{S}(y))$	$\rightarrow$	$\mathtt{A}(\mathtt{M}(x, y), x)$

Table 1.

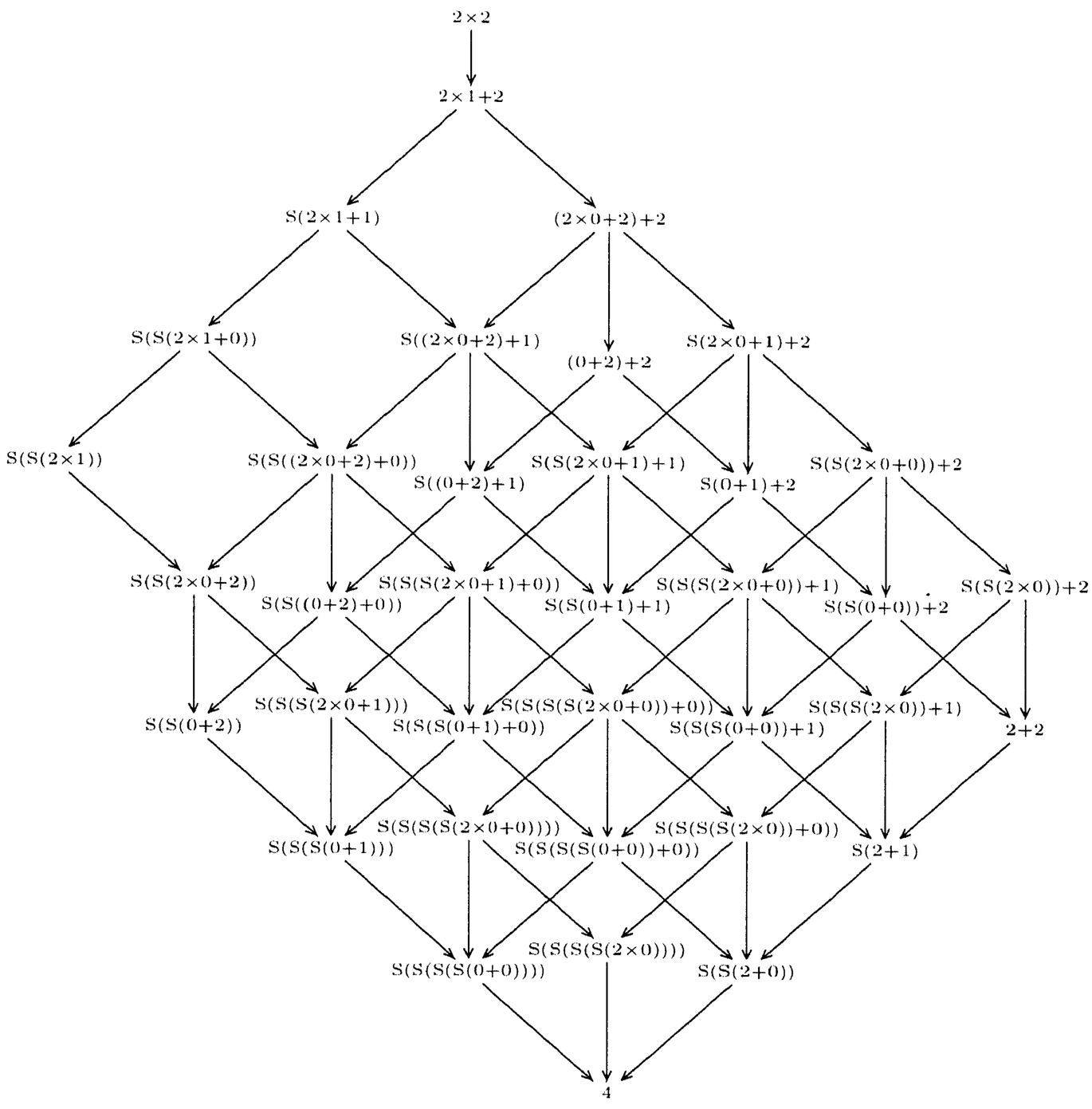


Figure 2.

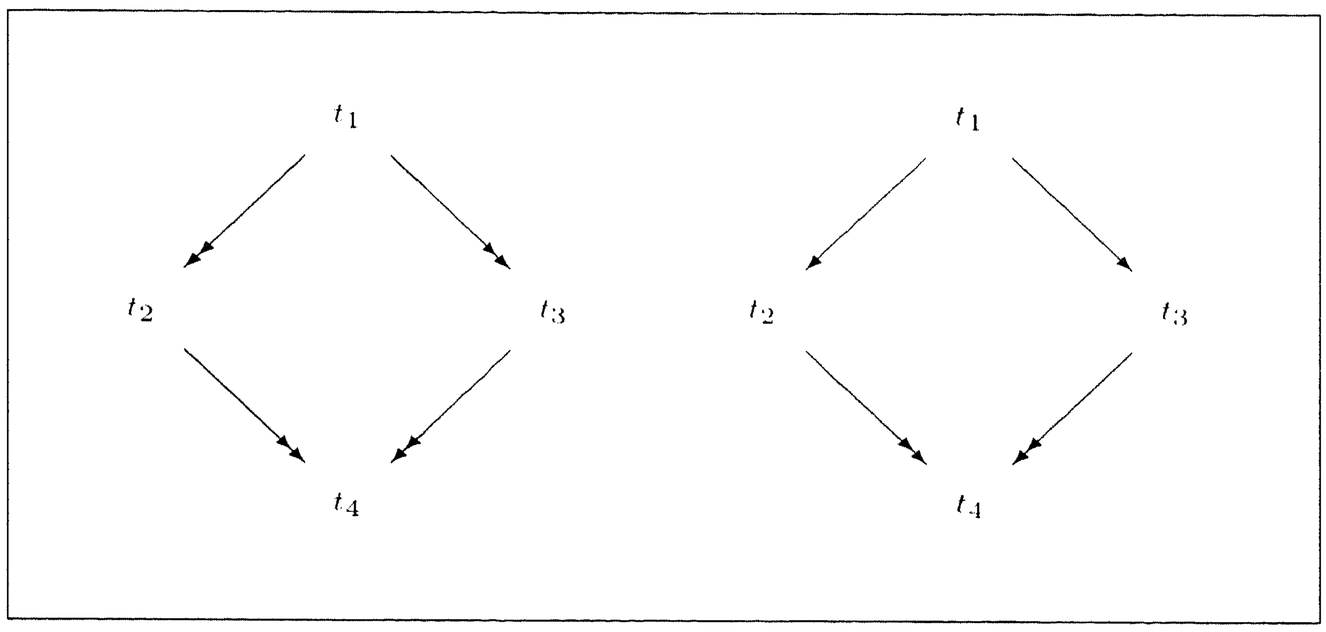


Figure 3. Confluence and weak confluence property.

$2 \times 2 = 4$ as displayed in figure 2, where the usual *infix* notation for `A` and `M` is employed. An inspection of figure 2, containing all possible computations or rewrite sequences of 2×2 to 4, is enough to wonder why all computations indeed yield the same final result or normal form. In other words why the rewrite system is *confluent* (see figure 3). There a double-headed arrow $\twoheadrightarrow$ denotes a sequence of rewrite steps of arbitrary length (possibly 0). Fortunately, the system for arithmetic is confluent as the rules have the technical property of *orthogonality* (they are independent of each other in the sense that applying one rule does not destroy the possible application of another rule.).

Figure 3 displays the confluence property, also called *Church-Rosser property*, which is next to the termination property the most fundamental property in rewriting. It guarantees the uniqueness of normal forms. The weaker property of *weak confluence* is useful to prove confluence, but actually not enough: turning back to the ARS in figure 1 we see that this rewrite system is weakly confluent, but not confluent. Every pair of divergent single *steps* can be joined again (by arbitrary long rewrite sequences), but not every pair of divergent rewrite *sequences* can be made to converge again. (E.g., the end points of the pair $b \to a$ and $b \to c \to d$ cannot come together any more.) Actually, the weak confluence property suggests that one can obtain confluence by repeatedly *tiling the plane* with tiles as in the figure for weak confluence. But this will not succeed always, as the tiling procedure might go on indefinitely, and diverge to yield some fractal-like picture as in figure 4.

But looking at this infinite *rewrite diagram*, it is easily and rightly conjectured that in the presence of the termination property we will have success with this tiling procedure.

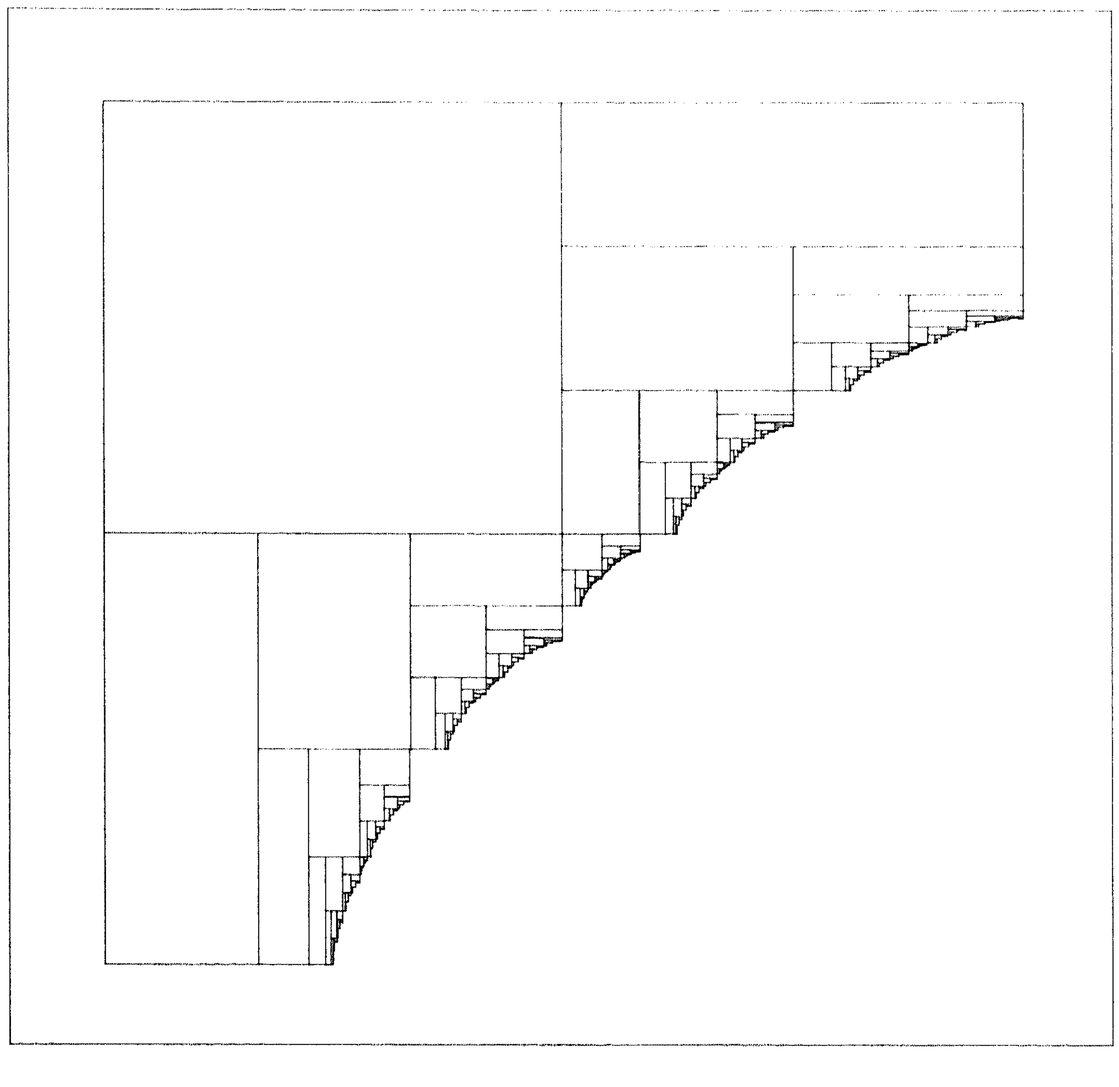


Figure 4.

5. COMBINATORY LOGIC

Actually, we do not need to devise special-purpose rewrite systems such as the one above in Table 1—there is a universal, general-purpose rewrite system, discovered in 1924 by M. Schönfinkel, called *Combinatory Logic*. Just as lambda calculus it is one of the perennial gems that mathematical logic has contributed to computer science. Combinatory Logic consists of the three rewrite rules in Table 2.

Here $\mathtt{S}, \mathtt{K}, \mathtt{I}$ are the basic constants and x, y, z are variables for terms. It is understood that a part of a term, built from $\mathtt{S}, \mathtt{K}, \mathtt{I}$ and matching the left-

r_1	$(((\mathtt{S} \cdot x) \cdot y) \cdot z)$	$\rightarrow$	$((x \cdot z) \cdot (y \cdot z))$
r_2	$((\mathtt{K} \cdot x) \cdot y)$	$\rightarrow$	x
r_3	$(\mathtt{I} \cdot x)$	$\rightarrow$	x

Table 2.

hand side of one of these rules, may be replaced by the corresponding right-hand side. The binary operator $\cdot$ is called *application*; often its notation is suppressed. Thus we have, e.g., the two step rewrite sequence

$$(((\mathtt{SK})\mathtt{I})\mathtt{I}) \to ((\mathtt{KI})(\mathtt{KI})) \to \mathtt{I}$$

which cannot be prolonged, since the final term I is irreducible (a normal form). Not all terms in CL can be rewritten to a normal form: for instance $((\mathtt{SI})\mathtt{I})((\mathtt{SI})\mathtt{I})$ cannot.

6. CONDITIONAL REWRITING

There are several ways to enhance, refine, or generalize first-order rewriting. One of them is *conditional rewriting*, an example of which is given in Table 3.

This system computes the greatest common divisor of natural numbers (generated by 0 and successor S) using the two conditional rewrite rules r_8 and r_9. The intended meaning of such conditional rewrite rules is that their application is only allowed if the condition to the right of $\Longleftarrow$ is fulfilled. Here a circularity is apparent: the conditions are stated themselves in terms of the rewrite relation $\to$ that they help to define. But a little bit of theory shows that this circularity is not harmful at all but quite innocent. Theory also has established (in an observation of J.A. Bergstra) that the conditional format is indeed strictly more powerful than the unconditional first-order scheme: some natural data types can be specified with a conditional rewrite system, but cannot without.

A different enhancement of first-order rewriting is to impose a certain order on the rewrite rules, with the intention that a rule which is higher in the order will be the preferred one to apply in case of choice. Such systems are called *priority rewrite systems*; their actual definition and semantics is

r_1	$0 < 0$	$\to$	$\mathtt{false}$		
r_2	$0 < \mathtt{S}(x)$	$\to$	$\mathtt{true}$		
r_3	$\mathtt{S}(x) < 0$	$\to$	$\mathtt{false}$		
r_4	$\mathtt{S}(x) < \mathtt{S}(y)$	$\to$	$x < y$		
r_5	$\mathtt{S}(x) - \mathtt{S}(y)$	$\to$	$x - y$		
r_6	$0 - x$	$\to$	0		
r_7	$x - 0$	$\to$	x		
r_8	$\mathtt{gcd}(x, y)$	$\to$	$\mathtt{gcd}(x - y, y)$	$\Longleftarrow$	$y < x \twoheadrightarrow \mathtt{true}$
r_9	$\mathtt{gcd}(x, y)$	$\to$	$\mathtt{gcd}(x, y - x)$	$\Longleftarrow$	$x < y \twoheadrightarrow \mathtt{false}$
r_{10}	$\mathtt{gcd}(x, x)$	$\to$	x		

Table 3.

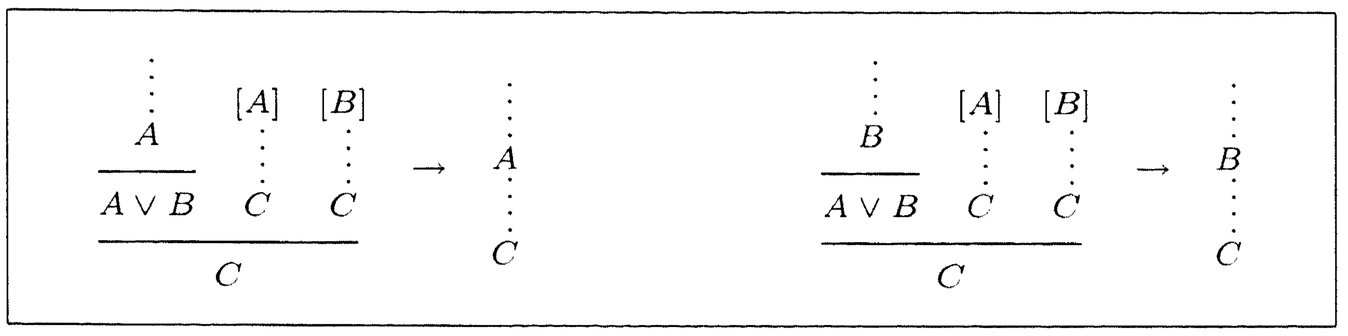


Figure 5.

technically difficult. An interesting and wide open area of investigation is given by the combination of the two last features, priorities and conditions.

7. HIGHER-ORDER REWRITING

A vast generalization is obtained when we go to *higher-order rewriting.* Here an essentially new feature is encountered (as compared to first-order rewriting): that of the bound variable, already well known in first-order predicate logic in quantified assertions as $\forall x\, \phi(x)$ (all x have property ϕ) and $\exists x\, \phi(x)$ (there exists an x with property ϕ).

The paradigm rewrite system of higher-order rewriting is another classical gem: lambda calculus. But higher-order rewriting has a wider scope and also includes rewrite systems appearing in Proof Theory such as the one in figure 5. These rewrite rules 'normalize' proofs in Natural Deduction by cutting away superfluous detours. The rules take in linear notation written in the formalism of *Combinatory Reduction Systems* (which constitutes one specific format for higher-order rewriting) the form displayed in Table 4. Here the alphabet of the Combinatory Reduction System consists of two unary function symbols `inl` and `inr` (for introduction of disjunction) and a ternary function symbol `el` (for elimination of disjunction).

8. INFINITARY REWRITING

For practical purposes one is often more interested in infinite objects than finite terms and their normal form. Such infinite objects can be given by a recursive ('circular') definition such as

$$\begin{array}{lcl} \texttt{el}(\texttt{inl}(Z), [x]Z_0(x), [y]Z_1(y)) & \rightarrow & Z_0(Z) \\ \texttt{el}(\texttt{inr}(Z), [x]Z_0(x), [y]Z_1(y)) & \rightarrow & Z_1(Z) \end{array}$$

Table 4.

```
#letrecones = 1 :: ones;;
```

which denotes the infinite sequence of ones, $111\cdots$, written in CAML, a modern functional programming language. (A note on syntax: # and :: denote the CAML prompt and list constructor, respectively, ;; represents the end of a sentence.)

The crucial manoeuvre to get infinite rewriting off the ground, is the formulation of the right notion of converging rewrite sequences. Namely, we have rewrite sequences which may take more than ω steps, where ω is the ordinal just after the natural numbers. So we need to know what is the *limit* of a rewrite sequence at limit ordinals λ. It turns out that the right notion of convergence towards a limit term is the one where not only an increasing part of the term is 'crystallized out', but also in this process the depth of the rewrite activity tends to infinity at every limit ordinal, ω, $\omega.2$, $\omega.3$, $\cdots$, $\omega^2, \cdots$. Figure 6 pictures this situation.

Infinitary rewriting is a point of view that can be applied to first-order rewriting, but also to the higher-order rewrite system of lambda calculus. Figure 7 displays a rewrite sequence of length $\omega+\omega$ involving infinite lambda terms. Infinitary lambda calculus has an important theoretical application: namely that of providing a semantics for cyclic lambda graph rewriting, discussed below.

9. TERM GRAPH REWRITING

In recent years attention has been given to a generalization of term rewriting called *graph rewriting*. The main idea, arising from the need for efficient implementation of term rewriting, is not to duplicate subterms when rewriting, but to share subterms by using pointers to just one copy. Also cyclic graphs are allowed. Thus the *fixed point combinator* `Y`, embodying the possibility

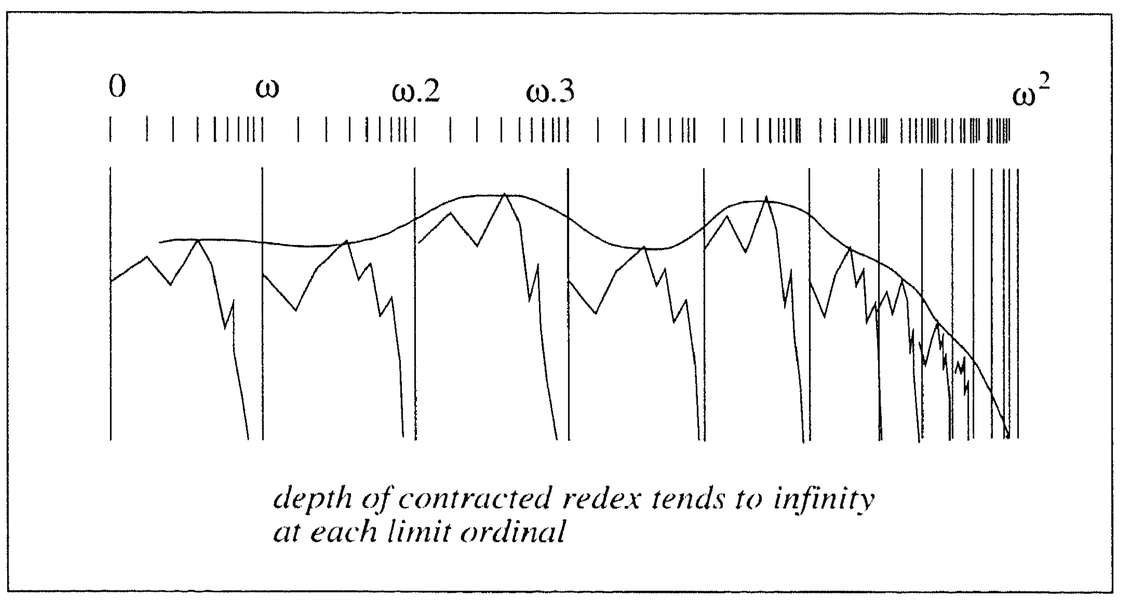


Figure 6.

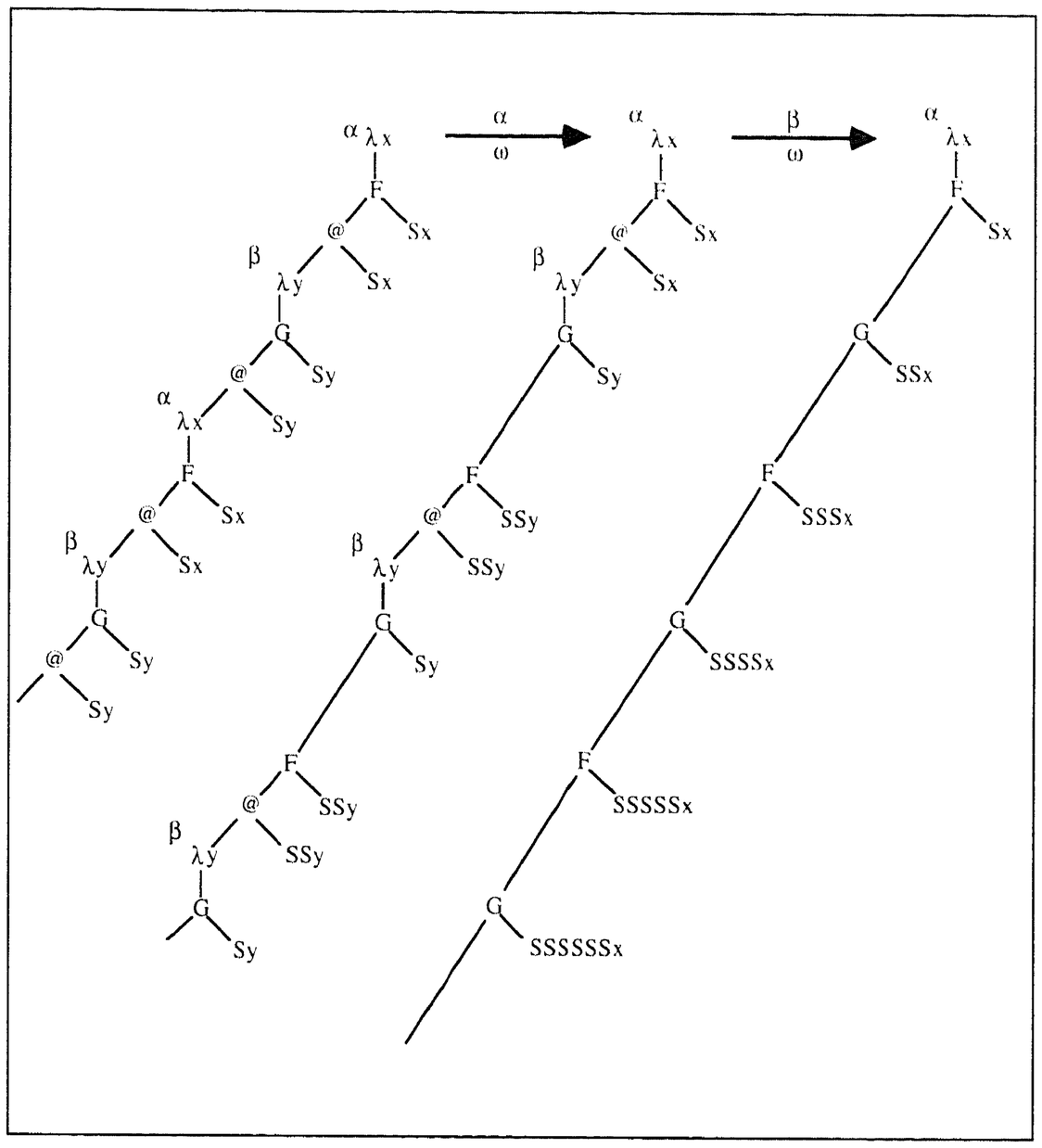


Figure 7.

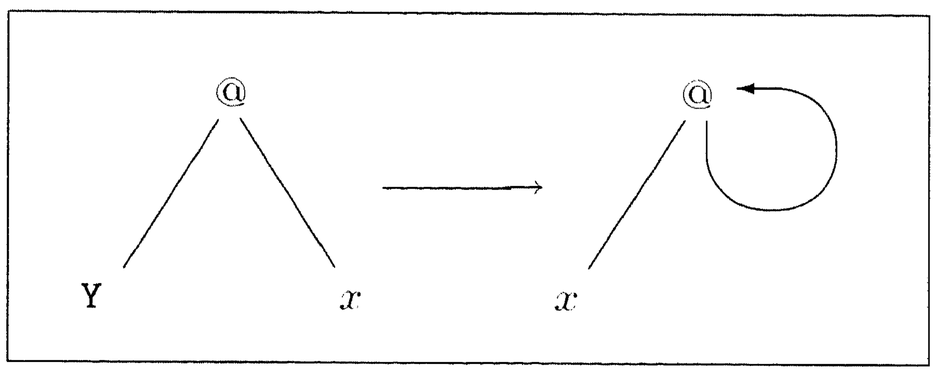
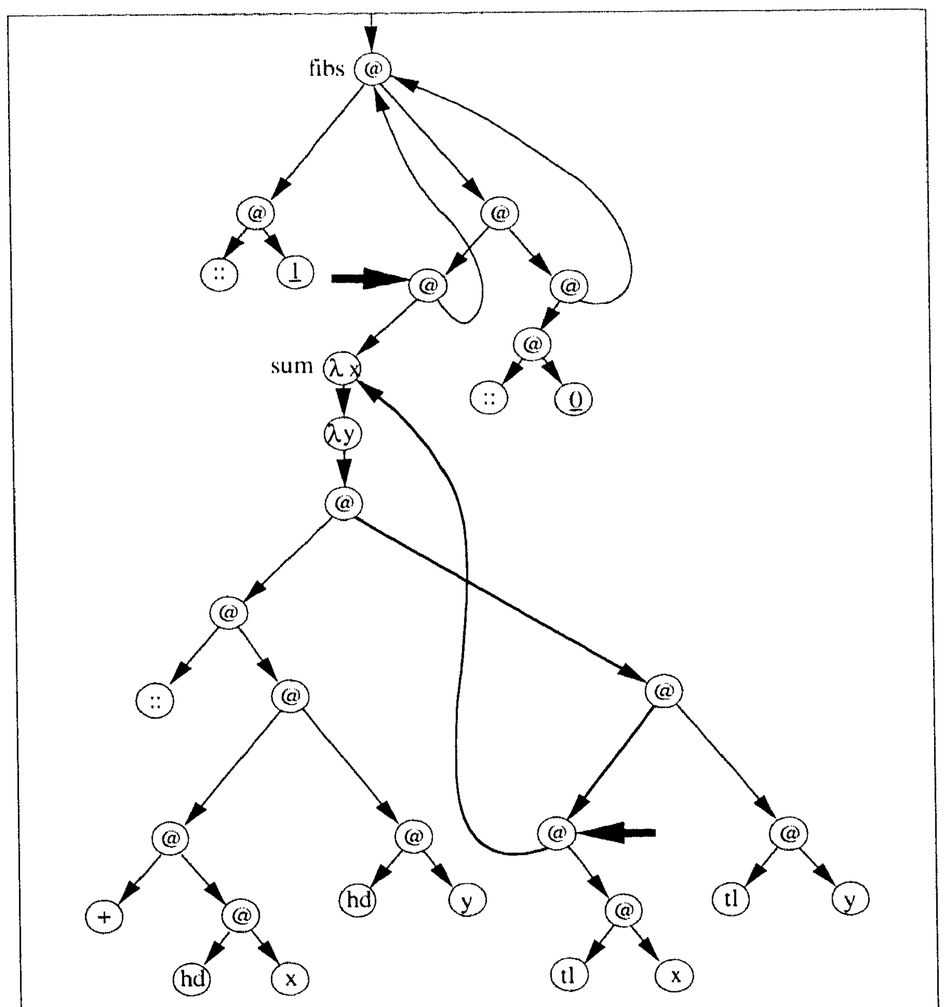


Figure 8.

of recursive definitions by means of its typical rewrite rule $\mathtt{Y} \cdot x \to x \cdot (\mathtt{Y} \cdot x)$, can be implemented in an elegant way as in figure 8. (Note that repeated application of the rewrite rule $\mathtt{Y} \cdot x \to x \cdot (\mathtt{Y} \cdot x)$ leads to the *infinite* term $x \cdot (x \cdot (x \cdot (x \cdot \ldots$, which is finitely presented by the right-hand side of the graph rewrite rule in the figure, where @ stands for $\cdot$.)

10. CYCLIC LAMBDA GRAPHS

Not only in the realm of first-order terms cycles are important, also for lambda calculus they constitute a useful new level of rewriting. While already occurring in the practice of functional programming, the theory of cyclic lambda calculus or as we prefer to say, lambda calculus with explicit recursion, is only in development since very recent years. As an example, consider the CAML specification of the sequence of Fibonacci numbers $1, 1, 2, 3, 5, 8, \cdots$.



Cyclic lambda graph for computing the sequence of Fibonacci numbers

Figure 9.

```
# let rec fibs = 1 :: sum fibs (0::fibs);;
# let rec sum = fun x y → (hd x + hd y) :: sum (tl x) (tl y) ;;
```

Graphically, this is a cyclic lambda graph as in figure 9. (The heavy arrows point to the roots of the two 'redexes' that are present in this graph ('redex = reducible expression'). An understanding of explicit recursion in lambda calculus serves to clarify the important programming concepts of 'let' and 'letrec'.

ACKNOWLEDGEMENTS

Many thanks to Z.M. Ariola, F. van Raamsdonk for helping out with the production of this paper in various ways, and to A. Middeldorp for kindly making figure 2 (from his Ph.D. thesis) available to us.

REFERENCES

1. Z.M. ARIOLA, J.W. KLOP (1994). Cyclic lambda graph rewriting. *Proc. Ninth Symposium on Logic in Computer Science (LICS'94)*, Paris, France, 416-425.
2. J.A. BERGSTRA, J.W. KLOP (1986). Conditional rewrite rules: confluence and termination. *J. Comput. System Sci. 32(3)*, 323-362.
3. J.R. KENNAWAY, J.W. KLOP, M.R. SLEEP, F.J. DE VRIES (1995). Transfinite reductions in orthogonal term rewriting systems. *Information and Computation 119(1)*, 18-38.
4. J.W. KLOP (1992). Term rewriting systems. S. ABRAMSKY, D. GABBAY, T. MAIBAUM (eds.). *Handbook of Logic in Computer Science*, volume II, Oxford University Press, 1-116.
5. J.W. KLOP, V.VAN OOSTROM, F.VAN RAAMSDONK (1993). Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science 121(1-2)*, 279-308. A Collection of Contributions in Honour of Corrado Böhm on the Occasion of his 70th Birthday, guest eds. M. Dezani-Ciancaglini, S. Ronchi Della Rocca and M. Venturini-Zilli.