Addressed Term Rewriting Systems

Frédéric Lang, Daniel Dougherty, Pierre Lescanne, and Kristoffer Rose

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme
46, Allée d'Italie, F–69364 Lyon Cedex 07, FRANCE
E-mail: `Frederic.Lang@ens-lyon.fr` *etc.*
URL: `http://www.ens-lyon.fr/LIP/PLUME/plume.us.html`
Phone: +33 (0)4 72 72 85 04 Fax: +33 (0)4 72 72 80 80

Abstract. We propose *Addressed Term Rewriting Systems* (ATRS) as a solution to the still-standing problem of finding a simple yet formally useful framework that can account for computation with sharing, cycles, and side effects. ATRS are characterised by the combination of three features: they

- work with terms where structural induction is useful,
- use a minimal "back-pointer" representation of cyclic data, and
- ensure a bounded complexity of rewrite steps by eliminating implicit pointer redirection.

In the paper we develop this and show how it is a very useful compromise between the less abstract term graph rewriting and the more abstract equational rewriting.

1 Introduction

Term Rewriting Systems (TRS, see [DJ90, Klo92, BN98]) are widely used to formalize, prototype, and prove software. However, they do not easily express some of the useful notions that one finds in software implementations, namely sharing, cycles, and (of course!) side effects:

Sharing. An essential notion to give an account of efficiency in the implementation of TRS. Indeed, while TRS rules such as the usual rule for squaring a number

$$\mathrm{square}(A) \;\to\; \mathrm{times}(A, A)$$

in principle copy a subterm of the input, real implementations optimize in only copying a *pointer* to this term. This not only saves memory but also makes it possible to *share future computations* on the referred subterm.

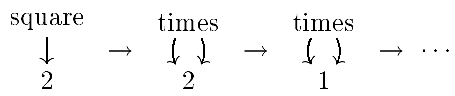
Sharing is, for example, essential in efficient implementations of lazy functional programming languages [PJ87], and the initial studies of sharing as "term graph rewriting" [BVEG$^+$87] were motivated by this application. However, one has to be careful. Consider the squaring TRS augmented with the rules

$$\begin{aligned} 2 &\;\to\; 1 \\ 1 &\;\to\; 2 \\ \mathrm{times}(1, 2) &\;\to\; \mathrm{ok} \end{aligned}$$

(these are silly but similar situations do arise in practice). This TRS permits the reduction

$$\mathrm{square}(2) \to \mathrm{times}(2,2) \to \mathrm{times}(1,2) \to \mathrm{ok}$$

If one uses sharing naively then the only possible reduction is the infinite



since it is impossible to "unshare" the copied node.

These issues have been studied for term graph rewriting systems (TGRS, see [Plu] for a survey), in particular the *adequacy* of TGRS to "implement" TRS [Plu93, KKSdV94, AKP97, EJ98]. However, the issue of guaranteeing *efficient* implementation has not been addressed until this work.

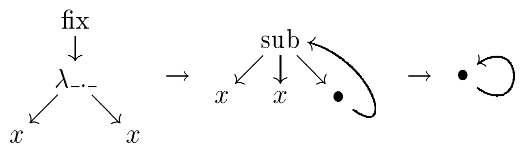
Cycles. These are also essential since one wants to deal with infinite data structures. A typical TRS rule that we would like to implement with a cycle is the notorious fixed point unfolding rule

$$\mathrm{fix}(\lambda x.E) \;\to\; \mathrm{sub}(E, x, \mathrm{fix}(\lambda x.E))$$

(with additional rules defining sub). This also illustrates the view that cycles are "self-sharing" or, as it is often calles, *vertical* sharing (in contrast to usual *horizontal* sharing). We would like to implement the above rule as something like

$$\mathrm{fix}(\lambda x.E) \;\to\; \mathrm{subst}(\;E\;,\;x\;,\;\bullet\;)$$

using a special "backpointer" indirection node. This creates more problems, however, in particular we have to deal specially with the "black hole" problem corresponding to the reduction



where the resulting graph does not really make sense Again, these are real problems that occur in connection with recursive equations in lazy functional programming languages (Haskell, Clean, *etc.*), or logic programming language (Prolog, Lambda Prolog, *etc.*).

The issues of cyclic structures has also been studied for TGRS [Ros92, AK96] and the problem is also related to "rational trees" [Cou83] which can be implemented in a finite setting as cyclic graphs but as in the general case none of these discuss the efficiency of the rewriting.

Side effects Side effects are a basic notion in many programming languages thus it is important to provide a formal framework for their description. For instance, in many object oriented programming languages, changing the value of a field in a shared object performs a side effect: the behaviour of other references to the same object is affected by the "non-local" update. Describing side effects in conjunction with parallel computing was one of the goals of the DACTL project [GKS89] but was never achieved, similarly, [AK96] conclude their definition of Equational Term Graph Rewriting by *"Furthermore, we intend to study the suitability of equational graph rewriting for expressing side-effect operations"*.

Side effects were, however, handled by [FF89] by introducing (what we call) *addresses*, to cite: *"we incorporate the store into the program component of the machine"*. This approach as well as the later "addressed explicit rewriting" [Ros96b] suffer from having a very low-level notion of reduction which thus makes generic algebraic reasoning difficult.

The main purpose of ATRS is to unify the two approaches to give an algebraic and general account of computation with a *memory state* while preserving the nice properties of simple term rewriting sufficiently to permit usual inductive reasoning. The idea is the same as in store-based models: the state of a memory is seen as a map from addresses to nodes, where each node carries a piece of information in the form of a *label* (for instance telling that the current node denotes the head of a list), and a vector of *successor addresses* (for instance, the vector containing the address of the next cell of the list) which make reference to the connected nodes, obtained through the state itself.

In other words, a state can be seen as a cyclic term graph. However, we eliminate the follozing problems with TGRS:

1. It is very hard to express *locality* of reduction with term graphs. Computers work by having a small "scratch area" in which they keep essentially a subterm of the entire problem. This can be seen as a small connected component of a term graph but we then still cannot observe the *inductive structure* of the considered subterm.
2. The notion of *reduction step* is not directly realizable as a single operation in the computer. The culprit is the *redirection phase* of [BVEG$^+$87]: this has an unbounded complexity in the sense that the number of pointers that have to be redirected depends on the context of the problem being solved.

With respect to the first issue, we note that, in the term graph model, memory addresses are modeled by "node identity". Thus, it is difficult to model a "move" of data in memory since one has to create new graph nodes corresponding to the new addresses occupied by the data. More precisely: when a piece p of a graph $g[p]$ is rewritten, it is completely replaced by a new piece p' on which connection to the graph context $g[\,]$ is only possible through identified nodes, *i.e.*, the root of p' (because it is treated as a special node) and the leaves of p' (because they are identified by variables), but not through other (internal) nodes of p'. In this sense TGRS models are "functional" because they cannot model non-functional issues such as side effects in that the rules cannot disregard the "term" structure and observe the term graphs as a flat sets of "address,constructor" pairs. This is the same problem as the traditional one has in denotational semantics where adding just the tiniest side effect forces one to use a completely sequential "store" model.

In this paper, we introduce addresses in a less obtrusive way: They are annotations on terms which play a rle in the rewriting process. Hence, we are allowed to express more interactions between parts of a graph, identified by their addresses. With this we can truly have "add-on side-effects" in the sense that only the parts of a specification that actually *breaks* locality needs to be specified as such. This also means that it is much easier to identify side effects in ATRS specifications, and in the paper we give a characterization of *side-effect free ATRS*.

With respect to the complexity of reduction steps, ATRS involve an explicit treatment of redirection by using *indirection nodes*. Implicit redirections *i.e.*, implicit change of pointers which pointed to a subgraph to point to another subgraph, as described in most papers on TGRS, is not permitted in ATRS. As a consequence we require a separate treatment of the so-called *collapsing rule* which resolves the "black hole" problem as described in [AK96]. In our setting, a black hole is a cyclic chain of indirection nodes which is observable just as in real implementations that print a "spine loop" error message when one is encountered. Hence we can state a similar result as Ariola and Klop: side effect free non overlapping ATRS are confluent, and for us this has a real operational significance.

With explicit indirection nodes to implement redirection we can claim that "counting rewrite steps" has a direct correspondance with the real complexity of implementations. Moreover, the way indirection nodes are handled in a real implementation can be dealt with in our framework, which allows to reason formally on the different choices of implementation one would be faced by, and to express properties of the consequences of these choices. For instance, one can imagine to prove that a system is *optimal* in its treatment of indirection nodes by never creating chains of indirections of length greater than one. Philosophically, this can be seen as the natural continuation of the wish of explicitness we have already exploited in our group, while working on lambda-calculi with explicit substitution (see [Les94, Ros96a] for surveys).

Finally, introducing addresses in the representation of a graph has some nice and useful consequences. For instance, it gives us the opportunity to represent graphs as trees (or terms) in a canonical way. Indeed, the addresses provide the required anotations, so that the graph (or the state) can be obtained from the addressed term, and vice versa. The advantages of this new representation of graphs upon the Barendregt and al. ones is that it allows a very rigorous definition of rewriting, while providing an intuitive correspondance between graphs and first order terms, graph rewriting rules and term rewriting rules.

Plan

In Section 2, we give some preliminary definitions, mainly to set our notations and recall the formal definition of term graphs. In Section 3, we define addressed terms, and show the correspondance between addressed terms and term graphs. In Section 4, we define graph matching and graph replacement on addressed terms, notions which will be used in Section 5, where we define the rewriting of addressed term, namely Addressed Term Rewriting Systems (ATRS), and show that the set of addressed terms is closed under rewriting. In Section 6, we give some examples of rewriting, in particular the example of a system which implements the setcar of Lisp *i.e.*, which handles side effects. In Section 7, we characterize side effects formally, and conjecture that side effect free non overlapping ATRS are confluent. We conclude in Section 8 and discuss further work.

We state many results in this paper, of which, for conciseness, we do not give the proofs, however, most of them are by induction over the term structure and thus easily checkable. All of them can be found in the full version of the paper [LDLR99].

2 Preliminaries

In this section, we give some definitions of structures which will be widely used in the following, namely sets and functions denoted as sets, and of term graphs.

Definition 1 (Sets, Functions, Tuples).

1. Sets are defined as usual: the empty set is denoted by $\varnothing$, union by $\cup$, and set membership by $\in$. S, e where S is a set, denotes the disjoint adjunction of e in S *i.e.*, requires that $e \notin S$. $\{e_1, \ldots, e_n\}$ denotes $S = \varnothing, e_1, e_2, \ldots, e_n$ which we will say is a set of size n and write $|S|$.
2. A (partial) function Φ from a set D to a set C is denoted by a set of pairs $(d; c)$ written $d \mapsto c$, such that $d \in D$, $c \in C$, and for all $d \in D$ there exists at most one c such that $d \mapsto c \in \Phi$. The predicate fun is defined on sets of pairs, where $\mathsf{fun}(S)$ is satisfied if and only if S is a function. The *domain* $\mathsf{dom}(\Phi)$ (resp. the *range* $\mathsf{rng}(\Phi)$) of the function Φ is the set $\{d \in D \mid \exists c, d \mapsto c \in \Phi\}$ (resp. $\{c \in C \mid \exists d, d \mapsto c \in \Phi\}$). The application of a function Φ to an argument $d \in \mathsf{dom}(\Phi)$ is written $\Phi(d)$ and equals c, where $d \mapsto c \in \Phi$. The notation $\Psi = \Phi, d \mapsto c$ means the extension of Φ into Ψ such that $\Psi(x) = \Phi(x)$ if $x \in \mathsf{dom}(\Phi)$ and $\Psi(d) = c$. It requires that $d \notin \mathsf{dom}(\Phi)$.
3. For a set S, S^n denotes the set of n-tuples with elements from S, written $\langle s_1; \ldots; s_n \rangle$. Formally an n-tuple is a map from $\{1, \ldots, n\}$ to S so $\langle s_1; \ldots; s_n \rangle(i) = s_i$ if $1 \leq i \leq n$. $S^0 = \varnothing$ is written $\langle\rangle$, and S^* denotes the union of the n-tuples for all natural n.

Term graphs [BVEG$^+$87, Plu] are rooted, directed, oriented, and cyclic graphs, whose nodes are labelled by symbols of a signature such that the number of successors of each node is given by the arity of its label.

Definition 2 (Term Graphs). Let Σ be a signature with a function arity : $\Sigma \mapsto \mathbb{N}$. A *term graph* over Σ is a quadruple $(N; r; \mathrm{lab}; \mathrm{succ})$ where

1. N is a finite set of *node identifiers*,

2. $r \in N$ is a distinguished node (identifier) called the *root*,
3. lab : $N \mapsto \Sigma$ maps each node to its *label*,
4. succ : $N \mapsto N^*$ maps each node to its *successor tuple*, and
5. it is required for all nodes $n \in N$ that $\mathrm{succ}(n) \in N^{\mathrm{arity}(\mathrm{lab}(n))}$.

In the following we call term graphs simply graphs.

3 Addressed Terms

An *addressed term* is a term decorated with addresses, which denotes a graph in a canonical way. Upon all terms decorated with addresses, which we call *addressed preterms*, some do not represent graphs. We first start to define those preterms, before giving formally the required conditions for such a preterm to be a term.

Definition 3 (Preterms).

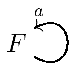
1. Let Σ be a term signature as in Definition 2, and $\bullet$ a special symbol of arity zero. Let $\mathcal{A}$ be an enumerable set of *addresses* denoted by $a, b, c, \ldots$, and $\mathcal{X}$ an enumerable set of *variables*, denoted by $x, y, z, \ldots$ An *addressed preterm* t over Σ is either:
   - a variable x;
   - or a constructed expression of the form $F^a(t_1, \ldots, t_n)$ where F is a symbol in Σ of arity $n \geq 0$, a is an address, and each t_i is an addressed preterm.
   - or the special addressed preterm $\bullet^a$, where a is an address.
2. The location of an addressed preterm t, denoted by $\mathsf{loc}(t)$, is defined by $\mathsf{loc}\big(F^a(t_1, \ldots, t_n)\big) = \mathsf{loc}(\bullet^a) = a$. It is not defined on variables.
3. The set of variables (resp. the set of addresses) of a term t is denoted by $\mathsf{var}(t)$ (resp. $\mathsf{addr}(t)$) and defined in the obvious way.

The notion of address is introduced to identify distinct nodes of the term as several occurrences of a unique node of the represented graph. Intuitively, if two subterms have a same address, this means that the subgraph they represent is shared *i.e.*, there are (at least) two paths from the root of the graph which go to this node (namely the paths of the occurrences of the address). It is clear that some coherence has to be preserved in the term such that it really does correspond to a graph. Specifically: if two subterms have the same address, then they indeed should represent a unique subgraph and thus have the same "contents". The coherence conditions will be given later.

However, to model cyclic memory structures with this "one occurrence per path" principle we would need infinitary trees. This idea has been pursued [KKSdV95] and one needs not only infinity of terms but also of rewrite sequences to capture that all the infinite number of copies may need be rewritten to reestablish a term that models a graph. The approach is a formally very clean way to achieve a model of cyclic structures but the loss of ordinary induction principles is a serious drawback when *using* the theory. The same holds, by the way, for the techniques used to reason about pointers in ML [Tof90, MT91] where one needs to use co-induction which is difficult to mix with "usual" term structure.

Our solution is the special symbol $\bullet$ called *back-pointer* used to give a representation of cyclic graphs as terms. Occurring in a term t, $\bullet^a$ denotes a reference to a node of t, which is located on the path going from the root of the term to itself, and has address a. On any cyclic path starting from the root in a cyclic graph, the first node visited twice is clearly identifiable and unique. This is this occurrence of the node which will appear as a back-pointer in the addressed term.

For instance, the preterm $F^a(\bullet^a)$ denotes the cyclic graph shown in the figure to the right (hence is a term), whose only node is given the address a. On the contrary, the addressed preterm $F^a(\bullet^b)$ does not represent any graph since b has no other occurrence on the path from the root to $\bullet^b$, hence does not point back.



While handling addressed terms, it is necessary to maintain some coherence in their representation. To this aim, we need two basic operations which are called *folding* and *unfolding*. Folding consists in replacing a subterm by a $\bullet$, corresponding to the introduction of a cycle in the addressed term, *e.g.*, after some rewriting step. Unfolding is dual to folding, since it performs the replacement of a $\bullet$ by an addressed term at the same location.

This is the price we pay[1] for insisting on a term structure: We have to unfold whenever a rewrite rule needs to observe the full subterm where it contains a cycle, because we cannot know by local observation in an addressed term what a $\bullet$ stands for.

The intuition is that folding and unfolding have no real operational meaning, but are used to keep the *representation* of a graph correct and unique. In fact, as we shall see later, the translation from an addressed term to a graph is invariant with respect to fold and unfold.

Definition 4 (Folding and Unfolding).

Folding. $\mathsf{fold}(a)(t)$, where t is a preterm, and a an address, is the *folding of preterms located at* a in t, defined as follows:

$$\mathsf{fold}(a)(x) = x \qquad\qquad \mathsf{fold}(a)(\bullet^b) = \bullet^b$$

$$\mathsf{fold}(a)\big(F^b(t_1,\dots,t_n)\big) = \begin{cases} \bullet^b & \text{if } a = b \\ F^b\big(\mathsf{fold}(a)(t_1),\dots,\mathsf{fold}(a)(t_n)\big) & \text{otherwise} \end{cases}$$

Unfolding. Let s and t be preterms, such that $\mathsf{loc}(s) = a$ (therefore defined), and a does not occur in t except as the address of $\bullet^a$. $\mathsf{unfold}(s)(t)$ is the

[1] A "conceptual" price, since in actual implementation it costs nothing.

unfolding of $\bullet^a$ *by* s *in* t defined as follows:

$$\mathsf{unfold}(s)(x) = x \qquad\qquad \mathsf{unfold}(s)(\bullet^b) = \begin{cases} s & \text{if } a = b \\ \bullet^b & \text{otherwise} \end{cases}$$

$$\mathsf{unfold}(s)\big(F^b(t_1, \ldots, t_m)\big) = F^b\big(\mathsf{unfold}(s')(t_1), \ldots, \mathsf{unfold}(s')(t_m)\big)$$
$$\text{where } s' = \mathsf{fold}(b)(s)$$

Thus addressed terms provide a representation of cyclic graphs as first order finite terms with annotations. In this, they resemble term graphs in the linear notation defined in [BVEG+87], with two major differences:

1. Every node has an address;
2. If in a graph there exists more than one (say n) acyclic path leading to the same node (i.e., to the same entry in a subgraph) this subgraph is duplicated n times in the external representation of the addressed term.

Therefore, the label and successors of each node are locally accessible, independently of the graph structure, the sharing information being given by the addresses. For instance, the graph which would be represented by the linear term $F(m : G(1,2), m)$, where m is the *name* of the node representing the root of $G(1,2)$, is represented by the addressed term $F^a(G^b(1^c, 2^d), G^b(1^c, 2^d))$ (uniquely, modulo an injective change of of addresses, called *address renaming*). The term $G(1,2)$ occurs only once in the linear notation, whereas it occurs twice in the addressed term, because it can be accessed following two distinct paths from the root. Of course, there is an exception for back pointers. However, for each of them, the information on the node it represents (*i.e.*, its label and successors) is present in its *vertical* context (*i.e.*, the context accessible by simply running to the root of the term). For instance, in the term $F^a(\bullet^a)$, the fact that the occurrence of $\bullet^a$ denotes a node whose label is F can be retrieved by looking up its context, namely $F^a(\Box)$.

This highlights the two advantages of the ATRS representation. First it makes structural induction on graphs possible, indeed induction on graphs boils down to induction on terms. As a consequence it makes type checking on cyclic graphs easily possible. Second, as we want to use these addressed terms in a rewriting framework, rewrite strategies will play a main rôle and this presentation of cyclic terms will make the description of strategies easy and precise.

We now proceed with the formal definition of *addressed terms* (called *terms* for short), *i.e.*, preterms which denote graphs.

Definition 5 (Term).

1. An *address context* Γ is a finite set of addresses.
2. A preterm t is a (addressed) term (or is *admissible*) in the address context Γ, if there exists Δ, a partial function from addresses to terms, such that $\Gamma \vdash t \triangleright$

Δ is derivable in the following system.

$$\frac{}{\Gamma \vdash x \rhd \varnothing} \qquad\qquad \frac{}{\Gamma, a \vdash \bullet^a \rhd \{a \mapsto \bullet^a\}}$$

$$\frac{\Gamma, a \vdash t_i \rhd \Delta_i}{\Gamma \vdash t \rhd \Delta} \quad \text{where} \quad \begin{array}{l} \Delta = \bigcup_{i=1}^{m}(\mathsf{unfold}(t) \circ \Delta_i) \cup \{a \mapsto t\}, \\ t = F^a(t_1, \ldots, t_m), \ \text{and } \mathsf{fun}(\Delta) \end{array}$$

In such a case, we say that t is admissible in the context Γ with *store* Δ. Note that it is clear from the rules, that if $\Gamma, a \vdash t_i \rhd \Delta_i$ succeeds, then t_i and Δ_i may only contain a as the address of $\bullet^a$ (provable by an easy induction on the structure of t). Therefore, the composition of $\mathsf{unfold}(t)$ with Δ_i in the third rule is well defined.

3. A preterm t is a (addressed) term (or is admissible) with store Δ if it is admissible in the empty context.

Note that if such a Δ exists for a given Γ, then it is unique and built deterministically, since the application of the rules is unambiguous and guided by the term syntax.

The store of a term is analogous to the equational system defining its graph, as defined in [AK96], and similar to the term graph expression (the latter will be formalised last in this section). The difference is that in our approach the system defining the graph and its subgraphs is somewhat *stratified*. In other word, instead of associating a flat term to each address, one associates an addressed (sub)term, which could itself be represented as an equational system. The condition for an addressed preterm to denote a graph is therefore simple: the system must be solvable *i.e.*, an address must be bound to at most one subsystem. This is stated in our setting by the requirement for the store to be a function. Another helpful analogy is of course to see the store as a real (stratified) store on a machine.

Example 1. The addressed term $H^c(F^b(G^a(\bullet^b)), G^b(F^a(\bullet^b)))$ has the store:

$$\{a \mapsto G^a(F^b(\bullet^a)), b \mapsto F^b(G^a(\bullet^b)), c \mapsto H^c(F^b(G^a(\bullet^b)), G^a(F^b(\bullet^a)))\}$$

One sees the following:

- Every address of t has an entry in Δ. This will be confirmed by Proposition 6.
- Every term of the range of the store is an admissible term, with a store reduced to its set of addresses. This will be confirmed by Proposition 7. Actually, $\Delta(a)$ is the term representing the graph accessible from the node located at a.

Proposition 6. *If $\Gamma \vdash t \rhd \Delta$, then* $\mathsf{addr}(t) = \mathsf{dom}(\Delta)$.

Proposition 7. *If $\Gamma \vdash t \rhd \Delta$, and $a \in \mathsf{addr}(t)$, then $\Gamma \vdash \Delta(a) \rhd \Phi$ with $\Phi \subseteq \Delta$.*

The following definition exploits the fact that, in an addressed term, in a given address context, there is a canonical term associated to each address, although there may be (syntactically) different terms which are located at this address. This term is given by the function Δ as follows.

Definition 8. Given a term t such that $\varnothing \vdash t \rhd \Delta$, and given $a \in \mathsf{addr}(t)$, $t\,@\,a$ is $\Delta(a)$. It is called the subterm of t at address a.

Corollary 9. *Let t be an admissible term and $a \in \mathsf{addr}(t)$. $t\,@\,a$ exists, is unique, and is admissible.*

It is clear that there is an isomorphism between addressed terms and graphs *i.e.*, from a ground and instantiated addressed term, one can build a unique graph, and from such a graph one can get back to the starting term (modulo a renaming of addresses, see Definition 14). The last issue treated here is the formalisation of this correspondance, specifically with the term graph notion of Definition 2.

Definition 10. Given a ground addressed term t and a term graph g over the same signature Σ. We say they *correspond*, written $t \gtrdot g$, if we can derive $\vdash t \gtrdot g, \varnothing$ with the following inference system:

$$\frac{}{\vdash \bullet^a \gtrdot (\varnothing; a; \varnothing; \varnothing), \{a\}} \qquad \frac{\vdash t_i \gtrdot (A_i; a_i; \mathrm{lab}_i; \mathrm{succ}_i), B_i \quad \mathsf{fun}(\mathrm{lab}) \quad \mathsf{fun}(\mathrm{succ})}{\vdash F^a(t_1, \ldots, t_n) \gtrdot (A; a; \mathrm{lab}; \mathrm{succ}), B}$$

$$\begin{aligned} \text{where } A &= \textstyle\bigcup_{1 \le i \le n} A_i \cup \{a\}, \\ B &= \textstyle\bigcup_{1 \le i \le n} B_i \setminus \{a\}, \\ \mathrm{lab} &= \textstyle\bigcup_{1 \le i \le n} \mathrm{lab}_i \cup \{a \mapsto F\}, \\ \text{and } \mathrm{succ} &= \textstyle\bigcup_{1 \le i \le n} \mathrm{succ}_i \cup \{a \mapsto \langle \mathsf{loc}(t_1), \ldots, \mathsf{loc}(t_n) \rangle\} \end{aligned}$$

Theorem 11. *For any signature Σ the relation $\gtrdot$ is a total function from the ground addressed terms to term graphs (over that signature).*

What this says is that we can map each addressed terms into a unique term graph and furthermore that we can model all possible term graphs with addressed terms. This justifies our claim that addressed terms are exactly the preterms that model graphs.

In the following, we only deal with admissible preterms (*i.e.*, terms), unless stated otherwise.

4 Matching and replacement of addressed terms

In this section, we define the *matching* of addressed terms, which corresponds to the matching of graphs. The idea is that an addressed term t_1 matches another (open) addressed term t_2, called the *pattern*, if there exists a graph homomorphism, and a substitution, from the graph denoted by t_2 to the graph denoted by t_1. Moreover, the matching operation builds and returns the homomorphism and the substitution as functions. We also define *replacement* which is an operation which simply expresses how a graph can be updated so that some of its nodes are changed. These two notions will play a major rôle in the definition of rewriting.

Definition 12 (Matching).

1. Functions from addresses to addresses are called *prehomomorphisms*[2]. Injective prehomomorphisms are called *renamings*. Functions which bind variables to preterms are called *substitutions*. A *matching* is a pair of a prehomomorphism α and a substitution σ, denoted by $\langle \alpha; \sigma \rangle$.
2. Let p and t be two terms. t *matches* p with *prematching* Π if there exists a matching $\langle \alpha; \sigma \rangle$ such that $\mathsf{match}\langle \Pi; p; t \rangle = \langle \alpha; \sigma \rangle$ can be obtained from the following system.

$$\frac{\mathsf{match}\langle \Pi, a \mapsto b; p_i; t'_i \rangle = \langle \alpha_i; \sigma_i \rangle \qquad \mathsf{fun}(\alpha \cup \{a \mapsto b\}) \qquad \mathsf{fun}(\sigma)}{\mathsf{match}\langle \Pi; F^a(p_1, \ldots, p_m); F^b(t_1, \ldots, t_m) \rangle = \langle \alpha \cup \{a \mapsto b\}; \sigma \rangle}$$

$$\text{where } \langle \alpha; \sigma \rangle = \langle \textstyle\bigcup_{i=1}^m \alpha_i; \bigcup_{i=1}^m \sigma_i \rangle, \quad t'_i = \mathsf{unfold}(F^b(t_1, \ldots, t_m))(t_i)$$

$$\frac{\mathsf{loc}(t) = b}{\mathsf{match}\langle \Pi, a \mapsto b; \bullet^a; t \rangle = \langle \{a \mapsto b\}; \varnothing \rangle} \qquad \frac{}{\mathsf{match}\langle \Pi; x; t \rangle = \langle \varnothing; \{x \mapsto t\} \rangle}$$

   Note that in the definition of $\mathsf{match}\langle \Pi; p; t \rangle$, the facts that p is a term in the context $\mathsf{dom}(\Pi)$, and that t is a term in the empty context, are invariant. We say that t *matches* p if t matches p with an empty prematching.
3. A matching $\langle \alpha; \sigma \rangle$ *fits* an admissible pattern p in an address context Γ if
   - $\mathsf{dom}(\alpha) \supseteq \Gamma \cup \mathsf{addr}(p)$ *i.e.*, every address of p or of its context has an image by α, and
   - $\mathsf{dom}(\sigma) \supseteq \mathsf{var}(p)$ *i.e.*, every variable of p has an image by σ, and
   - $\forall t \in \mathsf{rng}(\sigma), \exists \Delta_t$ such that
     - $\alpha(\Gamma) \vdash t \rhd \Delta_t$, where $\alpha(\Gamma) = \{\alpha(a) \mid a \in \Gamma\}$, and
     - $\mathsf{fun}(\bigcup_{t \in \mathsf{rng}(\sigma)} \Delta_t)$ *i.e.*, the terms of $\mathsf{rng}(\sigma)$ are, in some sense, *mutually coherent* in that they form together a coherent store.

Lemma 13. *If* $\mathsf{match}\langle \varnothing; p; t \rangle = \langle \alpha; \sigma \rangle$ *then* $\langle \alpha; \sigma \rangle$ *fits* p.

[2] Such functions serve to define homomorphisms on graphs, since they can add sharing in the graph, but never remove such sharing. However, the application of (the extension to terms of) a prehomomorphism to an addressed term does not always result in an addressed term, which is the reason why we do not call them *homomorphisms*. However, the prehomomorphisms computed by the match function will be real homomorphisms.

Definition 14 (Application of a matching). Let $\langle\alpha;\sigma\rangle$ be a matching fitting a term p. The application of $\langle\alpha;\sigma\rangle$ to p, denoted by $\mathsf{sub}\langle\alpha;\sigma\rangle(p)$, is defined as follows:

$$\mathsf{sub}\langle\alpha;\sigma\rangle(\bullet^a) = \bullet^{\alpha(a)} \qquad\qquad \mathsf{sub}\langle\alpha;\sigma\rangle(x) = \sigma(x)$$
$$\mathsf{sub}\langle\alpha;\sigma\rangle\big(F^a(p_1,\dots,p_m)\big) = F^{\alpha(a)}\big(\tau(p_1),\dots,\tau(p_m)\big)$$
$$\text{where } \tau = \mathsf{fold}(\alpha(a)) \circ \mathsf{sub}\langle\alpha;\sigma\rangle$$

The following proposition expresses that the matching computed by the match function is indeed a function allowing to compute the term from the pattern.

Proposition 15. *If* $\mathsf{match}\langle\Pi;p;t\rangle = \langle\alpha;\sigma\rangle$ *then* $\mathsf{sub}\langle\alpha;\sigma\rangle(p) = t$.

Lemma 16. *If* $\langle\alpha;\sigma\rangle$ *fits* p *then* $\mathsf{sub}\langle\alpha;\sigma\rangle(p)$ *is a term.*

Example 2. The term $F^c(G^d(\bullet^d), G^d(\bullet^d))$ matches $F^a(x, G^b(x))$ with

$$\langle\alpha;\sigma\rangle = \langle\{a \mapsto c, b \mapsto d\}; \{x \mapsto G^d(\bullet^d)\}\rangle.$$

We now define *replacement*. The function replace operates on terms. Given a term, it changes some of its subterms at given locations by other subterms with the same address.

Definition 17 (Replacement). In the following, a *replacement* Φ is a function from addresses to terms such that $\mathsf{loc}(\Phi(a)) = a$. $\mathsf{repl}(\Phi)(t)$ is defined on terms as follows:

$$\mathsf{repl}(\Phi)(x) = x \qquad\qquad \mathsf{repl}(\Phi)(\bullet^a) = \begin{cases} \Phi(a) & \text{if } a \in \mathsf{dom}(\Phi) \\ \bullet^a & \text{otherwise,} \end{cases}$$

$$\mathsf{repl}(\Phi)\big(F^a(t_1,\dots,t_m)\big) = \begin{cases} \Phi(a) & \text{if } a \in \mathsf{dom}(\Phi) \\ F^a\big(\mathsf{repl}(\Phi')(t_1),\dots,\mathsf{repl}(\Phi')(t_m)\big) & \text{otherwise,} \\ & \text{where } \Phi' = \mathsf{fold}(a) \circ \Phi \end{cases}$$

5 Addressed Term Rewriting Systems

We define in this section how rewriting is performed on addressed terms. Therefore, we give the definitions of rewriting rule, and sets of rewriting rules *i.e.*, ATRS.

Definition 18 (Rewriting Rule).

1. An addressed rewriting rule over Σ is a pair of addressed terms (l, r) over Σ, written $l \to r$, such that:
   - $\mathsf{loc}(l) = \mathsf{loc}(r)$ (same top address, therefore l and r are not variables);
   - $\mathsf{var}(r) \subseteq \mathsf{var}(l)$ (no creation of variables);

2. A term t *is a redex* for a rule $l \to r$ if there is a matching $\langle \alpha; \sigma \rangle$ such that $t = \mathsf{sub}\langle \alpha; \sigma \rangle(l)$. A term t *has a redex* if there exists an address $a \in \mathsf{addr}(t)$ such that $t\,@\,a$ is a redex.

Note that we do not set any restriction of linearity in addresses, and that l and r may be eventually cyclic. Note also that projection rules, such as $F^a(x) \to x$ or $F^a(G^b(x)) \to G^b(x)$ are not addressed rewriting rules since both sides do not share the same address. However, we propose to add to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\ \rceil$. Therefore, the TRS rule $F(x) \to x$ can be encoded by the ATRS rule $F^a(x) \to \lceil x \rceil^a$, and the TRS rule $F(G(x)) \to G(x)$ either by the rule $F^a(G^b(x)) \to G^a(x)$, which expresses the copy at address a of the label and successor addresses of the node located at address b, namely $G(x)$, or by the rule $F^a(G^b(x)) \to \lceil G^b(x) \rceil^a$, which expresses an indirection from the node located at address a to the node located at address b. The use of explicit indirection nodes is motivated by our wish to explicit the constraints that every rewriting step must be as close as possible to what happens in a real implementation. In particular the cost of the application of a rule must be almost constant in the implementation, so that the complexity of the rewriting and the complexity of the execution are tightened. Allowing projection rules would involve implicit redirections, whose cost depends on the number of references to the redirected node.

Example 3. $\mathrm{mult}^a(\mathrm{succ}^b(x), y) \to \mathrm{plus}^a(y, \mathrm{mult}^c(x, y))$ is an addressed rewriting rule. By its non right linearity, it expresses some sharing (y must be seen as a pointer). It creates a new node in the graph, whose address is denoted by c and which was not present in the left hand side. Moreover, unlike in classical graph rewriting, the node at address b is only dereferenced, but not deleted .

The following definition introduces *fresh renaming.* Informally, a fresh renaming is required when new nodes are introduced by rewriting step. It is important to give these new nodes addresses which are not already used. Fresh renaming captures this intuition.

Definition 19 (Fresh Renaming). Let $l \to r$ be an addressed rewriting rule, and t a term having a redex for this rule. A *fresh renaming* for $l \to r$ *w.r.t.* t is a renaming denoted by α_{fresh} such that:

– $\mathsf{dom}(\alpha_{\mathsf{fresh}}) = \mathsf{addr}(r) \setminus \mathsf{addr}(l)$ *i.e.*, the fresh renaming renames each newly introduced address, to avoid capture.
– $\mathsf{rng}(\alpha_{\mathsf{fresh}}) \cap \mathsf{addr}(t) = \varnothing$ *i.e.*, the chosen addresses are *fresh*, which means not present in t.

Lemma 20. *Let $l \to r$ be a rule. If $\langle \alpha; \sigma \rangle$ fits l* w.r.t. *a term t, and α_{fresh} is a fresh renaming for $l \to r$* w.r.t. *t, then $\langle \alpha \cup \alpha_{\mathsf{fresh}}; \sigma \rangle$ fits r* w.r.t. *t.*

Corollary 21. *Let $l \to r$ be a rule, and t a term. If $\langle \alpha; \sigma \rangle$ fits l, and α_{fresh} is a fresh renaming for $l \to r$* w.r.t. *t, then $\mathsf{sub}\langle \alpha \cup \alpha_{\mathsf{fresh}}; \sigma \rangle(r)$ is a term.*

At this point, we have given all the definitions needed to define rewriting.

Definition 22 (Rewriting). Let t be a term which we want to reduce at address a by rule $l \to r$. Proceed as follows:

1. Ensure $t \,@\, a$ is a redex *i.e.*, there is a matching $\langle \alpha; \sigma \rangle$, obtained by deriving
$$\mathsf{match}\langle \varnothing; l; t \,@\, a\rangle = \langle \alpha; \sigma \rangle$$
2. Compute α_{fresh}, a fresh renaming for $l \to r$ *w.r.t.* t.
3. Compute $s = \mathsf{sub}\langle \alpha \cup \alpha_{\mathsf{fresh}}; \sigma\rangle(r)$, and Φ such that $\varnothing \vdash s \rhd \Phi$ (which exists, according to Corollary 21).
4. The result of rewriting t by rule $l \to r$ at address a is $\mathsf{repl}(\Phi)(t)$.

Addressed term rewriting is performed in a similar way as term rewriting: first find a redex, then remind its context, then compute the result, then replace it in the previous context. An extra step is performed here, which is the creation of fresh addresses. In some sense, term rewriting can be seen as a special case of addressed term rewriting, where all addresses are distinct (no sharing and no side effects) and where there are no back-pointers (no cycles).

Definition 23 (Addressed Term Rewriting Systems). An Addressed Term Rewriting System (ATRS) over a signature Σ is a set of addressed rewriting rules over Σ.

The following corollary ensures that the set of addressed terms is closed under rewriting.

Corollary 24. *Let t be a term. If $t \to t'$ then t' is a term.*

6 Example of a side effect

The following shows how side effects are performed, by taking the famous example of setcar. To our knowledge, it is the first time the setcar is defined in the setting of graph rewriting.

Example 4. Consider the symbols nil of arity 0, and cons of arity 2, the constructors of lists. car, of arity 1 is the operator which returns the first element of a list, and setcar, of arity 2, the operator which affects a new value to the first element of the list, performing a side effect. We also have an infinite number of symbols denoted by $1, 2, \ldots, n, \ldots$, and the operation plus of arity 2. At last, we have the special symbol $\lceil\ \rceil$ of arity 1 denoting an indirection node. The rules for the definition of car and setcar are the following:

$$\mathrm{setcar}^a(\mathrm{cons}^b(y,z),x) \to \lceil \mathrm{cons}^b(x,z)\rceil^a \qquad \text{(Set-Car)}$$
$$\mathrm{setcar}^a(\lceil z\rceil^b,x) \to \mathrm{setcar}^a(z,x) \qquad \text{(Elim-Ind)}$$
$$\mathrm{car}^a(\mathrm{cons}^b(n^c,z)) \to n^a \qquad \text{(Car-}n\text{)}$$

We do not give the definition of plus since it is not the purpose of this example. Suppose it is built-in. Note that this system is infinite, since there is an infinity of rules (Car-n), namely one for each $n \in \mathbb{N}$. Take the term

$$t = \mathrm{plus}^e(\mathrm{car}^c(\mathrm{setcar}^a(\mathrm{cons}^b(1^f, \mathrm{nil}^h), 2^g)), \mathrm{car}^d(\mathrm{cons}^b(1^f, \mathrm{nil}^h)))$$

in which the list $\mathrm{cons}^b(1^f, \mathrm{nil}^h)$ appears twice, which means that it is shared. Starting by applying the rule for setcar, the reduction proceeds as follows:

$$\begin{aligned}
t &\to \mathrm{plus}^e(\mathrm{car}^c(\lceil \mathrm{cons}^b(2^g, \mathrm{nil}^h)\rceil^a), \mathrm{car}^d(\mathrm{cons}^b(2^g, \mathrm{nil}^h))) && \text{(Set-Car)}\\
&\to \mathrm{plus}^e(\mathrm{car}^c(\mathrm{cons}^b(2^g, \mathrm{nil}^h)), \mathrm{car}^d(\mathrm{cons}^b(2^g, \mathrm{nil}^h))) && \text{(Elim-Ind)}\\
&\to \mathrm{plus}^e(2^c, \mathrm{car}^d(\mathrm{cons}^b(2^g, \mathrm{nil}^h))) && \text{(Car)}\\
&\to \mathrm{plus}^e(2^c, 2^d) && \text{(Car)}\\
&\to 4^e && \text{(Plus)}
\end{aligned}$$

On the other hand, starting by applying the rule for car, the computation proceeds as follows:

$$\begin{aligned}
t &\to \mathrm{plus}^e(\mathrm{car}^c(\mathrm{setcar}^a(\mathrm{cons}^b(1^f, \mathrm{nil}^h), 2^g)), 1^d) && \text{(Car)}\\
&\to \mathrm{plus}^e(\mathrm{car}^c(\lceil \mathrm{cons}^b(2^g, \mathrm{nil}^h)\rceil^a), 1^d) && \text{(Set-Car)}\\
&\to \mathrm{plus}^e(\mathrm{car}^c(\mathrm{cons}^b(2^g, \mathrm{nil}^h)), 1^d) && \text{(Elim-Ind)}\\
&\to \mathrm{plus}^e(2^c, 1^d) && \text{(Car)}\\
&\to 3^e && \text{(Plus)}
\end{aligned}$$

The above example shows that ATRS may produce side effects. It is essential to be able to determine easily whether a system has side effects or not. For instance, a side effect appears clearly in the rule (Set-Car), since the term $\mathrm{cons}^b(y, z)$ becomes $\mathrm{cons}^b(x, z)$ whereas the rewriting takes place at address a in a term of shape $\mathrm{setcar}^a(\square, \square)$. Rules which do not host such transformations have no side effect.

7 Side effect free ATRS and Confluence

As shown in Example 4, non overlapping ATRS may be not confluent since they allow side effects. In this section, we characterize a subclass of the ATRSs which are conjectured confluent. We call them *side effect free non overlapping ATRS*.

Definition 25 (Overlapping, Side Effect).

1. Let $\tau_1 = l_1 \to r_1$ and $\tau_2 = l_2 \to r_2$ be two rules, not necessarily distinct. τ_1 *overlaps* with τ_2 if and only if:
   - there exists a term t, and an address a in l_1, such that t matches both $l_1 @ a$ and l_2,
   - if $\tau_1 = \tau_2$, then a is not the location of members of the rule.

   A system in which no rules overlap is called *non overlapping*.

2. A rule $l \to r$ has a side effect if and only if there exists an address a in $\mathsf{addr}(l) \cap \mathsf{addr}(r)$, distinct from the location of both members, such that $l \,@\, a$ is syntactically different from $r \,@\, a$. A rule or an ATRS with no side effects is called *side effect free.*

The name "side effect free" stems from the analogy with programming languages using a store: a rule which only modifies one place in the store (namely the location of the redex) does not have a side effect.

Theorem 26. *Side effect free non overlapping ATRS are confluent modulo address renaming.*

Proof. The conditions of side effect freeness and non overlapping are strong enough to ensure that the reduction of one redex may only eventually discard the other redex (in which case the fact it is discarded after being reduced or not is unimportant *w.r.t.* confluence), but never modify the part of the graph matched by the other redex.

8 Conclusion

We have defined in this paper the Addressed Term Rewriting Systems, which have a non empty intersection with the Term Graph Rewriting Systems: In particular, implicit redirections are not allowed in ATRS due to the constraint that both members of a rule must be located at the same address. The new feature of ATRS is that they give an account of side effects. We have shown that rewriting is a closed relation on the set of terms, and conjectured that side effect free non orthogonal ATRS are confluent. Moreover, we have given many examples illustrating the power of ATRS.

Further Work

We think that an interesting work would be to define *rewriting strategies.* Indeed, the term structure makes easy the definition of such strategies, since it is simply a function which, given a term, returns an address, namely the next address where reduction must take place. This principle was used in earlier work on rewriting with addresses [BRL96] to define weak reduction of the lambda-calculus, and show how the classical strategies of functional programming languages, namely call-by-value, call-by-name, and call-by-need, can be defined. Since ATRS is a restriction on addressed rewriting we look forward to develop the notion of address-defined strategies in this framework. In particular it is interesting to notice that strategies can be built independently of the calculus with side effects it may be associated to, whereas usually, systems with side effects and their strategies are tightly coupled. Similarly it would be nice if the notion would generalise to include the "logical rules" used in [Ros93].

In the same spirit, we would like to give an account of types, and typing properties, of ATRS. This has already started to be studied in [LLL99], in the setting of a lambda-calculus with objects, defined as an extension of the weak lambda-calculus with explicit substitution and addresses of [BRL96], and of the Lambda-Calculus of Objects of [FHM94]. We have tried to prove that this calculus, implementing mutable states, had the property of *Type Soundness* (via *Subject Reduction*). We have not yet succeeded because of imprecisions in our specification of ATRS at that time, which has led to this more formal paper.

At last, we think that ATRS provide a good language (for both a computer and a computer scientist) to express graph rewriting rules with side effects, which could lead to a better understanding of the implementations of programming languages.

Acknowledgements

The authors thank Zine-El-Abidine Benaissa [Ben97] who contributed to the elaboration of ATRS concepts, which helped to clarify many of the ideas of this paper. They also thank Luigi Liquori for useful dicussions and comments and for his delicious pasta.

References

[AK96] Z.M. Ariola and J.W. Klop. Equational term graph rewriting. *Acta Informatica*, 1996.

[AKP97] Z.M. Ariola, J. W. Klop, and D. Plump. Confluent rewriting of bisimilar term graphs. In *Proc. Fourth Workshop on Expressiveness in Concurrency*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.

[Ben97] Z.-E.-A. Benaissa. *Les calculs de substitutions explicites comme fondement de l'implantation des langages fonctionnels*. PhD thesis, Université Henri Poincaré – Nancy 1, 1997.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BRL96] Z.-E.-A. Benaissa, K.H. Rose, and P. Lescanne. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Proc. of PLILP*, number 1140 in Lecture Notes in Computer Science, pages 393–407. Springer-Verlag, 1996.

[BVEG+87] H. P. Barendregt, M. C. J. D. Van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *Proc. of PARLE '87*, number 259 in Lecture Notes in Computer Science, pages 141–158. Springer-Verlag, 1987.

[Cou83] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 244–320. Elsevier, 1990.

[EJ98] R. Echahed and J.-C. Janodet. Admissible Graph Rewriting and Narrowing. In *Proceedings of JICSLP'98*, 1998.

[FF89] M. Felleisen and D.P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69:243–287, 1989.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GKS89] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep. Final specification of Dactl. Report SYS–C88–11, University of East Anglia, Norwich, U.K., 1989.

[KKSdV94] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. On the Adequacy of Graph Rewriting for Simulating Term Rewriting. *ACM TOPLAS*, 16(3):493–523, 1994.

[KKSdV95] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

[Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Oxford University Press, 1992.

[LDLR99] F. Lang, D. Dougherty, P. Lescanne, and K.H. Rose. Addressed Term Rewriting Systems. Technical Report RR1999-30, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, 1999. Avalaible by ftp at `www.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-30.ps.Z`.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In *Proceedings of the 21st conference on Principles of Programming Languages*, pages 60–69, 1994.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A Framework for Defining Object Calculi. In *Proc. of Formal Methods (FM'99)*, Lecture Notes in Computer Science. Springer Verlag, 1999. To appear.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall International, 1987.

[Plu] D. Plump. Term graph rewriting. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific. To appear.

[Plu93] D. Plump. Collapsed tree rewriting: Completeness, confluence and modularity. In *Proc. Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 97–112. Springer Verlag, 1993.

[Ros92] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 of *LNCS*, pages 36–50, Pont-a-Mousson, France, July 1992. Springer-Verlag.

[Ros93] K.H. Rose. Graph-based operational semantics of a lazy functional language. In M. R. Sleep, M. J. Plasmeijer, and M. C. D. J. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 22, pages 303–316. John Wiley, 1993.

[Ros96a] K. H. Rose. Explicit substitution – tutorial & survey. Lecture Series LS–96–3, BRICS, Dept. of Computer Science, University of Aarhus, Ny Munkegade (bld.540), 8000 Aarhus C, Denmark, September 1996.

[Ros96b] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, Universitetsparken 1, DK-2100 København Ø, 1996. DIKU report 96/1.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

This article was typeset using the LaTeX macro package with the LLNCS2E class.