Compiling Collapsing Rules in Certain Constructor Systems

Sergio Antoy and Andy Jost

Computer Science Dept., Portland State University, Oregon, U.S.A.

antoy@cs.pdx.edu
andrew.jost@synopsys.com

Abstract. The implementation of functional logic languages by means of graph rewriting requires a special handling of collapsing rules. Recent advances about the notion of a needed step in some constructor systems offer a new approach to this problem. We present two results: a transformation of a certain class of constructor-based rewrite systems that eliminates collapsing rules, and a rewrite-like relation that takes advantage of the absence of collapsing rules. We formally state and prove the correctness of these results. When used together, these results simplify without any loss of efficiency an implementation of graph rewriting and consequently of functional logic computations.

1 Introduction

Functional logic programming [6, 18, 19] integrates the best features of the functional and the logic paradigms. For instance, demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming are combined with logic variables, constraint solving, and non-deterministic search from logic programming. Narrowing makes this combination seamless and enables encoding problems into programs in a style elegant, understandable, and easier to reason about [5].

Graph rewriting [9, 25, 27] is an approach to the implementation of functional and functional logic computations. The objects of a computation are *term graphs*, also referred to as *expressions*, i.e., singly rooted, acyclic graphs. For any graph t, $\mathcal{N}(t)$ is the set of nodes of t. A graph's *node* q has two attributes: a *label*, $\mathcal{L}(q)$, and a sequence of *successors*, $\mathcal{S}(q)$. The label and the successors abstract respectively a symbol of the signature of a rewrite system and the arguments to which the symbol's occurrence is applied in an expression. An implementation represents a node as a dynamic linked data structure holding a label and a sequence of pointers to other nodes. For technical convenience, graphs that differ only for a renaming of nodes are considered equal [15, 25].

A graph rewriting system, or *program*, is a set of *rules*, where a rule is a graph with two roots abstracting the left- and right-hand sides of the rule, respectively. Rules are *left linear* [12, Def. 1.4.1], i.e., the left-hand side is a tree. A consequence is that a variable occurs at most once in a left-hand side. A *step* of a computation of a host graph consists of three phases: (1) matching a rule left-hand side to a subgraph called the *redex*, (2) constructing the corresponding right-hand side called the redex's *contractum*, and (3) replacing the redex with its contractum. The signature from which the labels of the nodes are drawn is partitioned into *constructors* and *operations*. The left-hand side of a rule is a *pattern*, i.e., a graph rooted (by a node labeled) by an operation and every other node is labeled by either a variable or a constructor. A *constructor form*, or *value*, is a graph whose nodes are all labeled by constructors. A *head constructor form* is a graph rooted by a constructor.

Finding redexes in a graph according to some program is typically an expensive activity. However, this is not our case. For the inductively sequential graph rewriting systems (recalled below), a sound, complete and optimal strategy that finds redexes very efficiently is presented in [14, 15]. We consider a slightly more general class [3], that allows a well-behaved form of overlapping. The exact same strategy is applicable to our graphs with the only difference that some redexes have more than one contractum. In this case, in the spirit of functional logic programming, the contractum is chosen non-deterministically.

For example, the following rules, in Curry's syntax, define the function that computes the length of a list, where "[ ]" represents the empty list and $(x : xs)$ the list with head x and tail xs:

```
length [] = 0
length (x:xs) = 1 + length xs
```
(1)

A finite list is denoted $[x_1, \ldots x_n]$, where x_i, for any appropriate i, is an element of

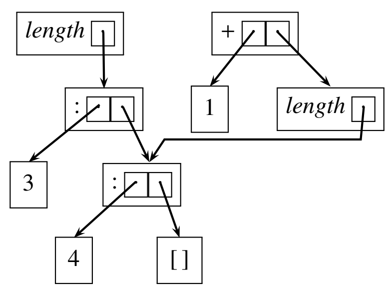

Fig. 1: Graph representation of the expression *length* [3, 4] (left) and its contractum 1 + *length* [4] (right). An outer box represents a node. Inside an outer box/node there is the label and a possibly null sequence of boxes representing references to the successors.

the list. The expression $t = length\,[3,4]$, which is a redex, is pictorially represented in Fig. 1. Conceptually, a rewrite step of t first constructs the contractum of t, $u = 1 + length\,[4]$, which is also shown in Fig. 1, and then redirects to u any reference to t (none occurs in the figure) because "t has become u." The redirection portion of a step [17] is a focus of our work.

Executing steps as described above would be naive and impractical. In fact, t can be a subexpression of a larger expression, called the context of t. The context of t may contain several references to t, i.e., the root of t is a successor of some nodes of its context. All these references should be tracked down and changed. This activity is potentially very expensive since a step is no longer a local operation, rather the entire context of t must be traversed. Our work deals with this specific aspect.

In this section, we recalled only the key concepts of graph rewriting needed to understand the problem and present our solution. Some familiarity with this framework is desirable. In Sect. 2 we recall two popular implementation techniques for graph rewriting. Since finding redexes in a host graph is easy and efficient in our framework, we focus only on the low-level details of nodes and pointers manipulation. In Sect. 3 we define the class of programs that we consider and recall recent results about properties of needed redexes in the class. These results are at the core of our technique. In Sect. 4 we define a program transformation that simplifies some aspects of executing those programs by graph rewriting. We state and prove our first correctness claim. In Sect. 5 we define a relation on graphs, called ripping, that produces results similar to rewriting, but is simpler to implement and more efficient to execute. We state and prove our second correctness claim. In Sect. 6 we statically quantify some effects of our technique on the performance of computations. In Sect. 8 we discuss related work and offer our conclusion.

2 Implementation Techniques

For the sake of efficiency, implementations of graph rewriting are usually "in-place." This means that in a step when the redex is replaced by its contractum, the context of the redex is re-used as the context of the contractum. This in-place rewriting still requires redirecting the pointers of the context pointing to the root of the redex. To avoid the cost of this operation, as discussed in the previous section, implementations of graph rewriting adopt special techniques.

The first technique is based on *indirection pointers* [23, Sec. 8.1]. Every node of an expression has an indirection pointer and is accessed only through this indirection pointer. The replacement of a redex t with its contractum u only needs redirecting to u the indirection pointer of t. Any reference within the context of t to the indirection pointer of t is unaffected. A step is a local operation using this technique, i.e., it does not require traversing the context of t. However, extra memory is allocated for every node of an expression and extra machine cycles are spent for every access to a node.

The second technique is based on destructive updates. In a step, the label and sequence of successors in the root of the redex are overwritten by the corresponding items that would be in the root of the contractum. We call such a step a *rip step* (re-labeling in place) and the technique, which we formalize in Sect. 5, *ripping*.

Ripping has several advantages over using indirection pointers—and one drawback. Among the advantages, references to the root of the redex do not need to be redirected to the root of the contractum; no indirection node is used; no node is allocated for the root of the contractum; and the root of the redex is reused rather than garbage collected. The drawback is that ripping may produce unintended results when a collapsing rule is applied. A *collapsing rule* is a rule whose right-hand side is a variable, which is called the *collapsing variable*. We show the problem on an example. Consider the following expression:

$$t = \texttt{(id x, id x) where x = 0 ? 1} \tag{2}$$

where *id* is the identity function:

```
id x = x                                              (3)
```

and "**?**" denotes the *choice* operation defined by the rules:

```
x ? y = x
x ? y = y                                             (4)
```

Contrary to popular functional programming languages, there is no textual order among the rules. Thus, the expression t **?** u, for any subexpressions t and u, non-deterministically rewrites to t or to u.

The meaning of the *where* clause in (2) is to introduce potentially shared nodes, where "shared" means having multiple predecessors. In the example, x is indeed shared.

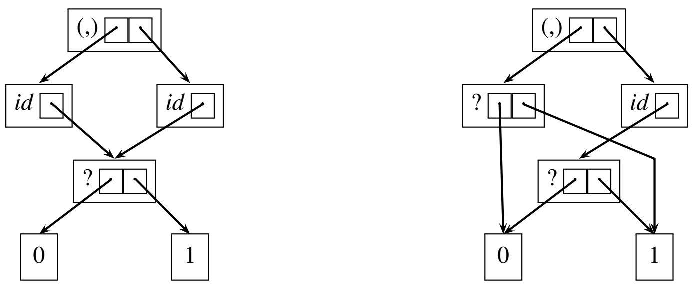

Fig. 2: The expression on the left-hand side has two values, $(0,0)$ and $(1,1)$. The expression on the right-hand side has 4 values, all possible pairs of zeros and ones.

The graph on the left-hand side of Fig. 2 pictorially shows t defined in program (2). This graph has two values, $(0,0)$ and $(1,1)$, resulting from each alternative of the *choice*. The graph on the right-hand side is obtained by a rip step of the redex in the first component of the pair. This graph has four values, all the pairs of zeros and ones. Two of these values, $(0,1)$ and $(1,0)$, are not intended. In a functional, hence deterministic setting, a graph has at most one value, thus, unintended values are not produced. However, the problem of duplicating portions of a computation still occurs and affects the efficiency of a computation rather than its input/output relation.

The problem we just showed is corrected by using a *forward node*. A forward node is a low-level device similar to an indirection pointer, but it is created only by steps applying collapsing rules, as opposed to systematically for every node, and explicitly to avoid the duplication of subexpressions. A program that manipulates graphs, e.g., for printing or evaluating them, must be aware of the possibility of encountering forward nodes and must be able to deal with them. During a computation, there is the danger of creating chains of forward nodes and the opportunity of compacting these chains to avoid the possibility of traversing them over and over.

In this paper, we propose a variation of the second technique, discussed in the previous page, based on destructive updates. Our variation does not require forward nodes. In short, we replace the collapsing rules of a program with non-collapsing rules in a way that does not change the "interesting" computations of the program. The motivation of our work is an implementation with destructive updates. Thus, we also formalize this implementation and discuss its correctness.

3 Detour on Need

Our overall approach to deal with collapsing rules is not to have any in a program. For example, consider the usual operation that concatenates two lists:

```
append [] ys = ys
append (x:xs) ys = x : append xs ys
```
(5)

The first rule is collapsing and *ys* is its collapsing variable. We recall that a *shallow* constructor expression is an expression of the form $c(x_1, \ldots c_n)$, where c is a constructor symbol of arity n and x_i is a fresh variable for every appropriate i. If we instantiate the collapsing variable with every shallow constructor expression of the variable's type, we obtain:

```
append [] [] = []
append [] (x:xs) = (x:xs)
append (x:xs) ys = x : append xs ys
```
(6)

where there are no collapsing rules. Programs (5) and (6) are similar. Given two lists, t_1 and t_2, if the expression *append* t_1 t_2 has a value according to (5), then it has the same value according to (6) and vice versa.

However, if *append* t_1 t_2 has no value according to (5) there is a difference. Consider the following non-terminating nullary operation:

```
loop = loop
```
(7)

The expression *append* [] *loop* is a redex according to (5), but it is not and it will never become a redex according to (6). In this section, we show that this difference is irrelevant for the execution of a program.

Our programs are modeled by a class of rewrite systems called overlapping inductively sequential [3]. *Inductive sequentiality* means that operations are defined by cases resembling those of a proof by structural induction. The rules of each operation can be organized in a hierarchical structure, called a *definitional tree* [2], that guides the evaluation strategy. *Overlapping*, in conjunction with the inductive sequentiality, means that if a redex is reduced by distinct rules, these rules have the same left-hand side. The epitome of an overlapping inductively sequential function is the *choice* operation defined in (4).

Every reducible expression t in the overlapping inductively sequential systems has a redex which is reduced by every computation of t to a value, a result that extends to a non-orthogonal class of systems the seminal result of [21]. A strategy that reduces only these redexes is optimal *modulo non-deterministic choices* [3].

A novel notion of need, more appropriate for constructor-based systems, was recently proposed in [7]. This notion depends only on the rules' left-hand side in a way that makes it applicable to the class of the overlapping inductively sequential systems that we just described.

Definition 1. [7] *Let t and u be operation-rooted expressions with u subexpression of t, we say that u is* needed *for t iff in any derivation of t to a head constructor form, u is derived to a head constructor form.*

Observe that u needs neither be a redex nor be a proper subexpression. In fact, u may be irreducible and t is a needed subexpression of itself. We abuse the word "needed" because our notion generalizes the definition of needed redex [21] as follows. The contrapositive formulation is Def. 1 more expressively captures this concept of need: t cannot be derived to a head constructor form, unless u is derived to a head constructor form.

The following statement establishes the connection between the classic formulation of need [21] and our formulation.

Lemma 1. [7] *Let $\mathcal{R}$ be an overlapping inductively sequential system. If u is both a needed (in the sense of [21]) subexpression of t and a redex, then u is a needed (in the sense of our Def. 1) redex of t, i.e., it is reduced to a head constructor form in any derivation of t to a head constructor form.*

From now on, "need" and "needed" will refer to the concept defined in Def. 1. The following immediate consequence of the above lemma is at the core of our technique.

Corollary 1. *Let $\mathcal{R}$ be an overlapping inductively sequential system. If t is a redex according to $\mathcal{R}$ needed for some context $C[]$, u is the contractum of t, and u is (still) operation-rooted, then u is needed for $C[]$ as well.*

This result justifies our claim that programs (5) and (6) are equivalent in practice. Let $t = \mathit{append}\ [\,]\ u$ be a needed expression, where u is an operation-rooted subexpression. Program (6) attempts to evaluate u for matching a rule of *append* to t. Program (5) does not. However, since t is a needed redex, u is its contractum, and u is operation-rooted, by Cor. 1, u is needed as well. Thus, program (5) will eventually attempt to evaluate u to a head constructor form as program (6). In other words, u is equally needed and evaluated by both programs.

4 Transformation

We define below a transformation that takes a rewrite system possibly containing collapsing rules and produces an equivalent rewrite system without collapsing rules. The precise meaning of the equivalence of input and output systems of the transformation is formalized by Th. 1.

Definition 2. *Let $\mathcal{R}$ be a constructor-based rewrite system. The* collapse-free variant *of $\mathcal{R}$, denoted $\mathcal{R}_u$, is defined as follows: for each rule R of $\mathcal{R}$, if R is not collapsing, then R is in $\mathcal{R}_u$. Otherwise, for every constructor symbol c of the signature of $\mathcal{R}$, R_c is in $\mathcal{R}_u$, where R_c is the instance of R obtained by instantiating the collapsing variable of R to a shallow constructor expression rooted by c. No other rule is in $\mathcal{R}_u$.*

Of course, in a typed system only well-typed instantiations of the collapsing variable are considered. For example, program (6) is the collapse-free variant of program (5).

Collapsing rules in which the collapsing variable is polymorphic give raise to a potentially large number of instantiations. In modern computers with gigabytes of core memory, the amount of memory for holding these instantiations should hardly be a problem. A rule in these instantiations is selected according to the root symbol of the rule left-hand side argument. This is an efficient operation executed in constant time, i.e., independently of the number of rules. A technique in which the instantiations of collapsing rules are not explicitly generated in the executable code, is discussed later.

Observe that for any program $\mathcal{R}$, $\mathcal{R}$ and its collapse-free variant $\mathcal{R}_u$ have the same signature. A sound, complete, and optimal strategy exists [3] for overlapping inductively sequential term rewriting systems. The same strategy is applicable to overlapping inductively sequential graph rewriting systems. Eventually, we would like to replace a program with its collapse-free variant. Thus, we are pleased to discover that the same strategy exists for the replacement program.

Lemma 2. *Let $\mathcal{R}$ be an overlapping inductively sequential system. Then, the collapse-free variant of $\mathcal{R}$, $\mathcal{R}_u$, is an overlapping inductively sequential system.*

Proof. We prove that every operation of $\mathcal{R}_u$ has a definitional tree, hence $\mathcal{R}_u$ is inductively sequential. The signatures of $\mathcal{R}$ and $\mathcal{R}_u$ are the same. If f is an operation of $\mathcal{R}_u$, then it is an operation of $\mathcal{R}$. Since $\mathcal{R}$ is inductively sequential, f has a definitional tree, say $\mathcal{T}$. If f has a collapsing rule $l \to r$, there is a *leaf* node L of $\mathcal{T}$ whose pattern π is equal to l modulo a renaming of nodes and variables. Let x be the collapsing variable of $l \to r$. We replace this *leaf* node of $\mathcal{T}$ with a branch node B that has the same pattern π, and x as the inductive variables. The children of B are leaves whose rules are all and only the rules of f instantiating $l \to r$ in $\mathcal{T}_u$ according to Def. 2. Hence f has a definitional tree in $\mathcal{R}_u$. □

The following result precisely states the equivalence between a program $\mathcal{R}$ and its collapse-free variant $\mathcal{R}_u$. The values of an expression e in $\mathcal{R}_u$ are all and only the values of e in $\mathcal{R}$.

Theorem 1. *Let $\mathcal{R}$ be an overlapping inductively sequential system and $\mathcal{R}_u$ its collapse-free variant. For all expressions t and s over the signature of $\mathcal{R}$ (and $\mathcal{R}_u$), with s head constructor form, $t \overset{*}{\to} s$ in $\mathcal{R}$ iff $t \overset{*}{\to} s$ in $\mathcal{R}_u$.*

Proof. The "if" direction is immediate. If $t \to t'$ in $\mathcal{R}_u$, then $t \to t'$ in $\mathcal{R}$, since every rule of $\mathcal{R}_u$ is an instance of a rule of $\mathcal{R}$. Hence, any computation in $\mathcal{R}_u$ is also a computation in $\mathcal{R}$. The "only if" direction is proved by strong induction on the number of collapsing rules applied in $A = t \overset{*}{\to} s$ in $\mathcal{R}$. The base case is immediate, since every non-collapsing rule of $\mathcal{R}$, by construction, is a rule of $\mathcal{R}_u$. For the induction case, consider the first step of A, say a, that applies a collapsing rule. We consider whether the match of the collapsing variable in step a is a head constructor form. Case *true*: the computation in $\mathcal{R}_u$ can make the same step and the claim holds by the induction hypothesis. Case *false*: let w be the match. Corollary 1 proves that w is needed, hence A must derive it to a head constructor form w'. We can re-arrange the steps of A [3, Lemma 20] (as in the Parallel

Moves Lemma) so that the derivation of w into w' occurs before step a of A. By the induction hypothesis, $w \to w'$ in $\mathcal{R}_u$. After re-arranging the steps of A, the residual of step a satisfies case *true*, and the claim holds. □

The previous result easily extends from head constructor forms to constructor forms.

Corollary 2. *Let $\mathcal{R}$ be an overlapping inductively sequential system and $\mathcal{R}_u$ its collapse-free variant. For all expressions t and s over the signature of $\mathcal{R}$ (and $\mathcal{R}_u$), with s constructor form, $t \overset{*}{\to} s$ in $\mathcal{R}$ iff $t \overset{*}{\to} s$ in $\mathcal{R}_u$.*

Proof. By induction on the length of a derivation using Theorem 1. □

Curry is a candidate for the application of our results, but some programs that could benefit from our technique cannot be entirely or directly modeled by rewrite systems because of the presence of built-in types. Program (2) makes this point. The collapse-free variant of (2) should contain an instance of the rule of *id* for every integer.

A solution to this problem is to avoid the explicit instantiation of collapsing rules, and instead to compile them slightly differently from non-collapsing rules. When a collapsing rule R is going to be applied to a redex, the match of the collapsing variable is checked. If the match, say t, is rooted by a constructor c, the application proceeds as if R were instantiated by mapping the collapsing variable to a shallow constructor expression rooted by c. Otherwise, t is evaluated in an attempt to obtain a head constructor form t'. If t' is obtained, the rule application proceeds again as described above. Otherwise, it must be that either the evaluation of t does not terminate or terminates in an operation-rooted expression. The latter is a failure of the entire computation, since t is needed. The same outcome, whether non-termination or failure, would be obtained by any implementation, since t must be evaluated to a head constructor form.

Evaluating an expression to obtain a head constructor form is an activity provided by many implementations. Hence, a major task for the adoption of our technique is already available in these implementations. For example, the Pakcs implementation [20] of Curry, which maps Curry source code to Prolog source code, defines a predicate, *hnf*, exactly for this task. The same is true for the Basic Scheme [8], which defines an abstract function, H, for this task and implements it in OCaml.

Some compilers of Curry, e.g. Pakcs [20], use a similar approach to encode polymorphic functions, such as Boolean and constrained equalities. These functions are applicable to instances of every algebraically defined and built-in typed. They could be defined by one rule for every constructor or value. Instead, the availability of a test for head constructor form and a procedure that evaluates an expression to head constructor form avoid the proliferation of rules.

5 Ripping

The proof of correctness of the previous section to some extent completes our work. Given a program $\mathcal{R}$ possibly containing collapsing rules, we transform it into a program $\mathcal{R}_u$ without collapsing rules. This allows us to compile $\mathcal{R}_u$ according to any desired scheme without concerns for collapsing rules. We are guaranteed that the values computed by $\mathcal{R}_u$ are all and only those computed by $\mathcal{R}$ and that they are obtainable with the same strategy and in the same number of steps. Furthermore, the proof of Theorem 1 implicitly shows that a computation to constructor form has the same length in the two systems.

Of course, there is the expectation that the scheme adopted to compile $\mathcal{R}_u$ is correct. The motivation of our work is to compile $\mathcal{R}_u$ for ripping. We are not aware of any proof of its correctness and, indeed, we have not even found a statement of it. In this section we address this issue.

We recall that given two graphs t and s, a (graph) *homomorphism* [15, 26] of t into s is a mapping $\sigma : \mathcal{N}(t) \to \mathcal{N}(s)$ that preserves roots and for nodes not labeled by a variable, labels and successors, i.e.,

1. $\sigma(\mathcal{R}\mathrm{oot}(t)) = \mathcal{R}\mathrm{oot}(s)$
2. $\mathcal{L}(\sigma(q)) = \mathcal{L}(q)$, for every node $q \in \mathcal{N}(t)$ with $\mathcal{L}(q) \in \Sigma$;
3. $\mathcal{S}(\sigma(q))_i = \sigma(\mathcal{S}(q)_i)$, for every node $q \in \mathcal{N}(t)$ and appropriate index i.

Let t be a graph, $l \to r$ a rewrite rule, q a node of t and $\sigma : l \to t|_q$ a homomorphism, i.e., q is the root of a redex of t. We call *ripping*, denoted “$\odot\!\!\to$” the binary relation on graphs defined as follows: Let p be the root of $\sigma(r)$. $t' = t + \sigma(r)$ except at node q for which, in t', $\mathcal{L}(q) = \mathcal{L}(p)$ and $\mathcal{S}(q) = \mathcal{S}(p)$. In other words, the label and successors of q, in t', are replaced by those of p. This update makes the need of pointer redirection, which occurs during the replacement phase of a rewrite step, unnecessary.

Ripping produces results different from rewriting. Consider again program (2). During the evaluation of t, the rule of *id* is applied to the first component of the pair. Since the rule is collapsing, the argument is evaluated to a head constructor form. The result is non-deterministic, thus let us suppose that 0 is produced (if 1 were produced, the reasoning would be identical). The entire expression at this point is pictorially represented in the left-hand side of Fig. 3.

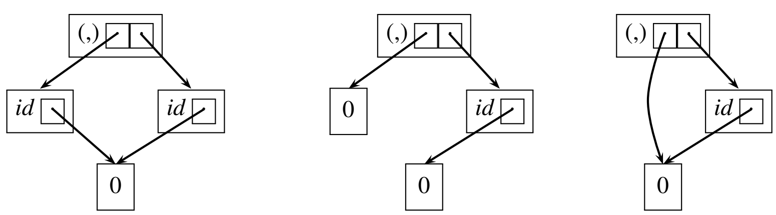

Fig. 3: The second graph is obtained from the first graph with a *rip* step, the technique formalized in this paper. The third graph is obtained from the first graph with a rewrite step.

The second graph of Fig. 3 shows the result of a rip step where the redex is the first component of the pair. The result is a graph with two nodes labeled by zero. We remark that no new node is created by this step, rather the root of the redex has been re-labeled with the label of the root of the contractum. The third graph is obtained by applying the same rewrite step to the first graph. We introduce the following concept to precisely characterize the significant differences between these graphs.

Definition 3. *Given two graphs t and s, t is an* adequate representation *of s iff there exists a homomorphism σ of t into s such that, for all distinct nodes p and q of t, if $\sigma(p) = \sigma(q)$, then the label of p (and hence of q) is a constructor symbol. We call such homomorphism an* adequate homomorphism*.*

For example, the second graph of Fig. 3 is an adequate representation of the third graph.

Observe that the match of the left-hand side of a rule to a redex is an adequate homomorphism since rules are left linear and that the composition of adequate homomorphisms is an adequate homomorphism. The diagram in Fig. 4 pictorially represents Lemma 3, where the vertical arrows stand for adequate homomorphisms.

$$\begin{tikzcd} t \arrow[r, "\odot\!\!\rightarrow" description, phantom] \arrow[d] & t' \arrow[d] \\ s \arrow[r] & s' \end{tikzcd}$$

Fig. 4

Lemma 3. *Let $\mathcal{R}$ be an overlapping inductively sequential system and $\mathcal{R}_u$ its collapse-free variant. Let t and s be graphs over the signature of $\mathcal{R}$ with t an adequate representation of s. Then, $t \odot\!\!\rightarrow t'$ in $\mathcal{R}_u$ (a rip step) for some t' iff $s \rightarrow s'$ in $\mathcal{R}_u$ (a rewrite step) for some s', where t' is an adequate representation of s'.*

Proof. Preliminarily, observe that the set of nodes of t labeled by an operation is in a bijection with the set of nodes of s labeled by an operation. Furthermore, if a graph g is an adequate representation of a graph h, and l is the left-hand side of a rewrite rule, then l matches g iff l matches h. Thus, for every step of t there is corresponding step of s, with the same rule, and vice versa.

Assuming we apply the same rule at corresponding nodes of t and s, we constructively prove the existence of an adequate homomorphism of t' into s'. Let's partition the nodes of t' into 3 classes: (1) the root of the redex, (2) the remaining nodes of t' that are also in t, and (3) the nodes created by the step, which originate from the nodes of the rule's right-hand side which are not labeled by a variable. A node in class (2) is also in t, thus it is mapped to make the diagram of Fig. 4 commutative. A node in class (3) is also in s, thus it is mapped to make the diagram of Fig. 4 commutative. The node, say q, in class (1) is mapped from a node in t, that is mapped to the root of the redex in s. Let p the root of the contractum of this redex. Thus, map q to p. This define a homomorphism which is adequate. □

The following result shows that ripping and rewriting compute the same values of an expression modulo an adequate representation.

Theorem 2. *Let $\mathcal{R}$ be an overlapping inductively sequential system and $\mathcal{R}_u$ its collapse-free variant. Let t and s be graphs over the signature of $\mathcal{R}$ with s a constructor form. If s is a value of t by rewriting in $\mathcal{R}_u$, then there exists an s' that is a value of t by ripping in $\mathcal{R}_u$ and s' is an adequate representation of s. If s' is a value of t by ripping in $\mathcal{R}_u$, then there exists an s that is a value of t by rewriting t in $\mathcal{R}_u$ and s' is an adequate representation of s.*

Proof. By induction on the length of a derivation. □

The combination of Th. 1 and Th. 2 shows that the evaluation of an expression by graph rewriting can also be obtained by ripping, in-place rewriting with re-labeling, which appears simpler and more efficient than other alternatives. This technique is simpler and faster when the rule being applied is not a collapsing rule. Our work shows that this is possible for every system in the class that we consider.

A computation in $\mathcal{R}_u$ executed by rewriting has a corresponding computation executed by ripping. We regard these two computation as the same. For every step of one computation, there is a step of the other computation that applies the same rule at a node that we regard as the same because in the hosting graphs there is a bijection between the nodes labeled by operations. The results of the two computations, that have nodes labeled by constructors only, may not be isomorphic graphs. However, they are equal both when printed as (tree) terms, because they are bisimilar [11], and when printed in fully collapsed form[1] [10, 26], because one is an adequate representation of the other.

6 Performance

The major contribution of our work is not a speedup of computations or a reduction of both static and dynamic memory consumption, though they all do occur in some degree, but a simplification of the compiler architecture—forward nodes, and the machinery to handle them, can be entirely eliminated at nearly no cost.

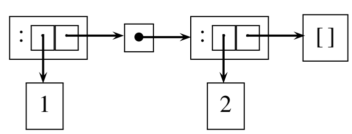

Fig. 4: The evaluation of *append* [1] [2] produces a list containing a forward node represented by the large black dot in the above diagram.

We begin our performance analysis with an example. Consider a program that concatenates some lists and computes the length of the result. For concreteness, we choose very simple lists, i.e., the program computes *length*(*append* [1] [2]). The rules of *length* and *append* were given in (1) and (5) respectively. The value of *append* [1] [2], say L, computed *without* the use of our technique is shown in Fig. 4. The large black dot represents the forward node created when the first rule of (5) is applied. The same value computed *with* our technique, is equal to L except that the forward node is absent. List L may never be entirely present in memory because operation *length* consumes portions of L as soon as operation *append* constructs portions of L due to the lazy evaluation strategy, but the order of evaluation does not affect our reasoning.

The execution time of each program is too short to be reliably measured with ordinary tools. As far as memory consumption is concerned, our technique saves the allocation and the traversal of the forward node. There is a similar program that instead

[1] The word "collapse" is overloaded in graph parlance. In this context, its refers to a relation on graphs defined in the cited reference.

of constructing a list of two elements separated by a single forward node it constructs a similar list with an arbitrarily long chain of forward nodes. Computing the length of this list takes an arbitrarily long time. More relevant is that the implementation of *length* must be prepared to encounter forward nodes. Hence, extra instructions are executed to check for their presence. When a forward node is encountered, extra instructions are executed to reach the node that the forward node points to. Thus, the object code of *length* is longer, is more complicated, takes longer to execute, and allocates extra dynamic memory.

Quantifying the practical effects of these differences is impractical. The average speed up of our technique and the savings in memory consumption depend on the programs used for a benchmark. And of course, programs with long chains of forward node are less frequent. A static analysis provides more precise information that, however, is more difficult to relate to execution times or memory consumption.

1. Without our technique, every time a collapsing rule is applied, a forward node is allocated and initialized. By contrast, our technique executes the same step with an instantiated rule. Therefore, the node corresponding to the collapsing variable is pattern matched and the content of the root of the redex is re-assigned.
2. Without our technique, every time a node is pattern matched, a test must be performed to check whether the node is a forward node. In the affirmative case, the node pointed to by the forward node must be fetched and pattern matched again. The fetched node could be a forward node again. By contrast, our technique avoids the test, and never has to fetch a second node.

7 Narrowing

Functional logic programs compute with unknown information which is abstracted by logic (also called *free*) unbound variables. A free variable is bound during a computation if and when without the binding the computation could not continue. The combination of binding some variables and making a rewrite step is called *narrowing*. Narrowing supports a simple and elegant programming style [5] unique to the functional logic paradigm.

For a contrived example, consider again the rule of (5) and the expression $t = append\ v\ []$, where v is an unbound free variable. No rule can be applied to t. To compute the value of t, v is bound to either [] or $(x : xs)$, non-deterministically, where x and xs are fresh unbound free variables. For example, if v is bound to [], the value of t is []. By contrast, consider the expression $s = append\ []\ v$, where v is again an unbound free variable. In this case, s is rewritten to v, where v is unaffected by the step. Variable v might be bound later depending on the context in which it occurs.

During the execution of a program, we store the bindings of free variables in an array called the *bind-table*. A variable is internally represented as an index in the bind-table array. The k-th entry in the array, holds the binding, if any, of the variable represented by k. A conventional value marks unbound variables. Any node standing for a variable is labeled by the same distinguished symbol, which we denote "*free*". In addition, in a node standing for a variable v, we store the index of v in the bind-table.

Regarding the integration of free variables with our technique, the only relevant question is what happens when, during the application of a collapsing rule, the *collapsing* variable is bound to a *free* variable. The answers is that we simply treat the free variable as if it were a head constructor form. I.e., the step replaces the content of the root of the redex with the content of the root of the replacement, in this case the node representing the free variable.

Graph rewriting stipulates that, for each variable v, in any expression there is at most one node labeled by v [15, 25]. Our approach violates this stipulation, but only in appearance. The index k of a node with label *free* is immutable. The binding, if any, indexed by k is in the bind-table. Thus, there is invariably one and only one binding of any variable regardless of the number of nodes standing for that variable. The claims leading to the correctness of our technique, Th. 2, carry over to narrowing with no significant changes. We only need a minimal extension to the notion of adequate representation. Referring to the notation of Def. 3, if $\sigma(p) = \sigma(q)$, then the label of p (and hence of q) is either a constructor symbol or *free*, and when the label is *free*, the indexes in p and q are the same.

8 Discussion and Related Work

Graph rewriting is a viable mean for the implementation of functional and functional logic languages that has lead to the discovery and development of optimal strategies [4]. Transformations of rewrite systems for compilation purposes are described in [16, 22]. The specialization of rules through the instantiations of collapsing variables is typical of partial evaluation [1]. Our goal differs from those of the above techniques. Our transformation is specialized in that its only purpose is to eliminate collapsing rules. Its merit is in the property that, for the class of systems that we consider, which is perfectly suited for functional logic programming, every computation to a value in a system with collapsing rules can be executed, with the same effort, in a system without collapsing rules. An implementation of rewriting without collapsing rules is easier to code and faster to execute. We have not found any work close enough to ours for a direct comparison.

Literature on the implementation of graph rewriting abounds. With respect to our work, papers fall into either of two groups, graph reduction machines [13, 24], or some specialized aspects of rewriting [23]. Our implementation of ripping as rewriting is theoretical in that we do not address data structures, register allocation, bit use for tags, and similar. Its merit is to make the pointer redirection phase of a rewrite step effortless in a concrete implementation. We have not found any description of this technique or claim of its correctness.

To our knowledge, this is the first paper addressing collapsing rules in conjunction with narrowing.

9 Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249. Michael Hanus provided valuable comments on a preliminary version of this paper.

References

1. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
2. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at `http://cs.pdx.edu/~antoy/homepage/publications/alp97/full.pdf`.
4. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
5. S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522, May 2010.
6. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85, April 2010.
7. S. Antoy and A. Jost. Are needed redexes really needed? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 61–71, New York, NY, USA, 2013. ACM.
8. S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31, Kobe, Japan, May 2012. Springer LNCS 7294.
9. Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *FUNDAMENTA INFORMATICAE*, 26, 1996.
10. Zena M. Ariola, Jan Willem Klop, and Detlef Plump. Bisimilarity in term graph rewriting. *Information and Computation*, 156(12):2 – 24, 2000.
11. 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 *PARLE Parallel Architectures and Languages Europe*, volume LNCS 259, pages 141–158, Eindhoven, 1987. Springer-Verlag. also technical report SYS-C87-01.
12. M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.
13. T. J.W. Clarke, P. J.S. Gladstone, C. D. MacLean, and A. C. Norman. Skim - the s, k, i reduction machine. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 128–135, New York, NY, USA, 1980. ACM.
14. R. Echahed. Inductively sequential term-graph rewrite systems. In *Graph Transformations, 4th International Conference (ICGT 2008)*, pages 84–98, Leicester, UK, 2008. Springer, LNCS 5214.
15. R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at `ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz`.

16. W. Fokkink and J. van de Pol. Simulation as a correct transformation of rewrite systems. In *In Proceedings of 22nd Symposium on Mathematical Foundations of Computer Science, LNCS 1295*, pages 249–258. Springer, 1997.
17. J. R. W. Glauert, R. Kennaway, G. A. Papadopoulos, and M. R. Sleep. Dactl: an experimental graph rewriting language. *J. Prog. Lang.*, 5(1):85–108, 1997.
18. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
19. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
20. M. Hanus, editor. *PAKCS 1.11.4: The Portland Aachen Kiel Curry System*, 2014. Available at `http://www.informatik.uni-kiel.de/~pakcs`.
21. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, I. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*, pages 395–414. MIT Press, Cambridge, MA, 1991.
22. J. F. T. Kamperman and H. R. Walters. Simulating TRSs by minimal TRSs a simple, efficient, and correct compilation technique. Technical Report CS-R9605, CWI, 1996.
23. J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors, *Term Graph Rewriting Theory and Practice*, pages 157–169. J. Wiley & Sons, Chichester, UK, 1993.
24. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.
25. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
26. Detlef Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51:277–289, 2001.
27. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term Graph Rewriting Theory and Practice*. J. Wiley & Sons, Chichester, UK, 1993.