

7-13-2015

## Compiling Collapsing Rules in Certain Constructor Systems

Sergio Antoy

Portland State University, antoys@pdx.edu

Andy Jost

Synopsys

Let us know how access to this document benefits you.

Follow this and additional works at: [https://pdxscholar.library.pdx.edu/compsci\\_fac](https://pdxscholar.library.pdx.edu/compsci_fac)

 Part of the [Computer Engineering Commons](#), [Computer Sciences Commons](#), and the [Electrical and Computer Engineering Commons](#)

---

### Citation Details

Antoy, Sergio and Jost, Andy, "Compiling Collapsing Rules in Certain Constructor Systems" (2015). *Computer Science Faculty Publications and Presentations*. 137.

[https://pdxscholar.library.pdx.edu/compsci\\_fac/137](https://pdxscholar.library.pdx.edu/compsci_fac/137)

This Post-Print is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. For more information, please contact [pdxscholar@pdx.edu](mailto:pdxscholar@pdx.edu).



39 replacing the redex with its contractum. The signature from which the labels of the  
 40 nodes are drawn is partitioned into *constructors* and *operations*. The left-hand side of a  
 41 rule is a *pattern*, i.e., a graph rooted (by a node labeled) by an operation and every other  
 42 node is labeled by either a variable or a constructor. A *constructor form*, or *value*, is a  
 43 graph whose nodes are all labeled by constructors. A *head constructor form* is a graph  
 44 rooted by a constructor.

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

53 For example, the following rules, in Curry’s syntax, define the function that com-  
 54 puts the length of a list, where “[ ]” represents the empty list and  $(x : xs)$  the list with  
 55 head  $x$  and tail  $xs$ :

$$\begin{aligned}
 \text{length } [] &= 0 \\
 \text{length } (x:xs) &= 1 + \text{length } xs
 \end{aligned}
 \tag{1}$$

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

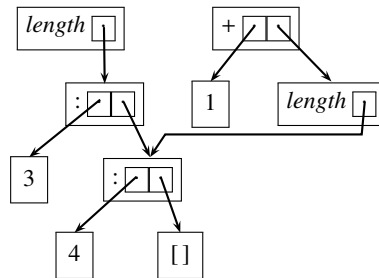


Fig. 1: Graph representation of the expression `length [3, 4]` (left) and its contractum `1 + length [4]` (right). An outer box represents a node. Inside an outer box/node there is the label and a possibly null sequence of boxes representing references to the successors.

57 the list. The expression  $t = \text{length } [3, 4]$ , which is a redex, is pictorially represented  
 58 in Fig. 1. Conceptually, a rewrite step of  $t$  first constructs the contractum of  $t$ ,  $u =$   
 59  $1 + \text{length } [4]$ , which is also shown in Fig. 1, and then redirects to  $u$  any reference to  $t$   
 60 (none occurs in the figure) because “ $t$  has become  $u$ .” The redirection portion of a step  
 61 [17] is a focus of our work.

63 Executing steps as described above would be naive and impractical. In fact,  $t$  can be  
 64 a subexpression of a larger expression, called the context of  $t$ . The context of  $t$  may con-  
 65 tain several references to  $t$ , i.e., the root of  $t$  is a successor of some nodes of its context.  
 66 All these references should be tracked down and changed. This activity is potentially

67 very expensive since a step is no longer a local operation, rather the entire context of  $t$   
68 must be traversed. Our work deals with this specific aspect.

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

## 83 2 Implementation Techniques

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

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

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

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

$$109 \quad t = (\text{id } \mathbf{x}, \text{id } \mathbf{x}) \text{ where } \mathbf{x} = \mathbf{0} ? \mathbf{1} \quad (2)$$

110 where  $id$  is the identity function:

$$111 \quad id \ x = x \quad (3)$$

112 and “?” denotes the *choice* operation defined by the rules:

$$113 \quad \begin{aligned} \mathbf{x} \ ? \ \mathbf{y} &= \mathbf{x} \\ \mathbf{x} \ ? \ \mathbf{y} &= \mathbf{y} \end{aligned} \quad (4)$$

114 Contrary to popular functional programming languages, there is no textual order among  
 115 the rules. Thus, the expression  $t \ ? \ u$ , for any subexpressions  $t$  and  $u$ , non-determinis-  
 116 tically rewrites to  $t$  or to  $u$ .

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

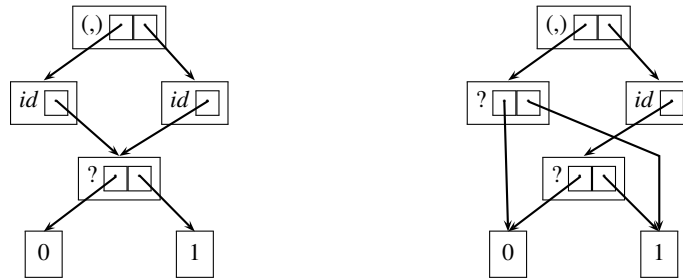


Fig. 2: The expression on the left-hand side has two values, (0, 0) and (1, 1). The expression on the right-hand side has 4 values, all possible pairs of zeros and ones.

119  
 120 The graph on the left-hand side of Fig. 2 pictorially shows  $t$  defined in program  
 121 (2). This graph has two values, (0, 0) and (1, 1), resulting from each alternative of the  
 122 *choice*. The graph on the right-hand side is obtained by a rip step of the redex in the  
 123 first component of the pair. This graph has four values, all the pairs of zeros and ones.  
 124 Two of these values, (0, 1) and (1, 0), are not intended. In a functional, hence determinis-  
 125 tic setting, a graph has at most one value, thus, unintended values are not produced.  
 126 However, the problem of duplicating portions of a computation still occurs and affects  
 127 the efficiency of a computation rather than its input/output relation.

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

136 In this paper, we propose a variation of the second technique, discussed in the pre-  
 137 vious page, based on destructive updates. Our variation does not require forward nodes.  
 138 In short, we replace the collapsing rules of a program with non-collapsing rules in a way

139 that does not change the “interesting” computations of the program. The motivation of  
 140 our work is an implementation with destructive updates. Thus, we also formalize this  
 141 implementation and discuss its correctness.

### 142 3 Detour on Need

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

$$145 \begin{aligned} \text{append } [] \text{ } ys &= ys \\ \text{append } (x:xs) \text{ } ys &= x : \text{append } xs \text{ } ys \end{aligned} \quad (5)$$

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

$$151 \begin{aligned} \text{append } [] \text{ } [] &= [] \\ \text{append } [] \text{ } (x:xs) &= (x:xs) \\ \text{append } (x:xs) \text{ } ys &= x : \text{append } xs \text{ } ys \end{aligned} \quad (6)$$

152 where there are no collapsing rules. Programs (5) and (6) are similar. Given two lists,  $t_1$   
 153 and  $t_2$ , if the expression  $\text{append } t_1 \text{ } t_2$  has a value according to (5), then it has the same  
 154 value according to (6) and vice versa.

155 However, if  $\text{append } t_1 \text{ } t_2$  has no value according to (5) there is a difference. Consider  
 156 the following non-terminating nullary operation:

$$157 \quad \text{loop} = \text{loop} \quad (7)$$

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

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

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

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

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

180 Observe that  $u$  needs neither be a redex nor be a proper subexpression. In fact,  $u$  may  
 181 be irreducible and  $t$  is a needed subexpression of itself. We abuse the word “needed”  
 182 because our notion generalizes the definition of needed redex [21] as follows. The con-  
 183 trapositive formulation is Def. 1 more expressively captures this concept of need:  $t$   
 184 cannot be derived to a head constructor form, unless  $u$  is derived to a head constructor  
 185 form.

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

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

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

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

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

## 204 4 Transformation

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

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

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

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

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

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

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

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

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

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



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

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

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

263 *Proof.* By induction on the length of a derivation using Theorem 1.  $\square$

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

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

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

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

## 291 5 Ripping

292 The proof of correctness of the previous section to some extent completes our work.  
293 Given a program  $\mathcal{R}$  possibly containing collapsing rules, we transform it into a program  
294  $\mathcal{R}_u$  without collapsing rules. This allows us to compile  $\mathcal{R}_u$  according to any desired  
295 scheme without concerns for collapsing rules. We are guaranteed that the values com-  
296 puted by  $\mathcal{R}_u$  are all and only those computed by  $\mathcal{R}$  and that they are obtainable with the

297 same strategy and in the same number of steps. Furthermore, the proof of Theorem 1  
 298 implicitly shows that a computation to constructor form has the same length in the two  
 299 systems.

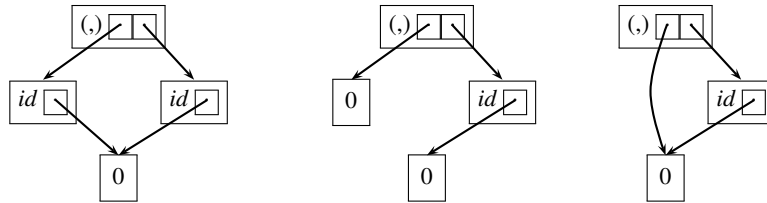
300 Of course, there is the expectation that the scheme adopted to compile  $\mathcal{R}_u$  is correct.  
 301 The motivation of our work is to compile  $\mathcal{R}_u$  for ripping. We are not aware of any proof  
 302 of its correctness and, indeed, we have not even found a statement of it. In this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

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

- 307 1.  $\sigma(\text{Root}(t)) = \text{Root}(s)$
- 308 2.  $\mathcal{L}(\sigma(q)) = \mathcal{L}(q)$ , for every node  $q \in \mathcal{N}(t)$  with  $\mathcal{L}(q) \in \Sigma$ ;
- 309 3.  $\mathcal{S}(\sigma(q))_i = \sigma(\mathcal{S}(q)_i)$ , for every node  $q \in \mathcal{N}(t)$  and appropriate index  $i$ .

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

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



321 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.

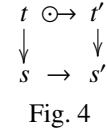
322 The second graph of Fig. 3 shows the result of a rip step where the redex is the first  
 323 component of the pair. The result is a graph with two nodes labeled by zero. We remark  
 324 that no new node is created by this step, rather the root of the redex has been re-labeled

325 with the label of the root of the contractum. The third graph is obtained by applying  
 326 the same rewrite step to the first graph. We introduce the following concept to precisely  
 327 characterize the significant differences between these graphs.

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

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

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



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

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

347 Assuming we apply the same rule at corresponding nodes of  $t$  and  $s$ , we construc-

348 tively prove the existence of an adequate homomorphism of  $t'$  into  $s'$ . Let's partition  
 349 the nodes of  $t'$  into 3 classes: (1) the root of the redex, (2) the remaining nodes of  $t'$  that  
 350 are also in  $t$ , and (3) the nodes created by the step, which originate from the nodes of  
 351 the rule's right-hand side which are not labeled by a variable. A node in class (2) is also  
 352 in  $t$ , thus it is mapped to make the diagram of Fig. 4 commutative. A node in class (3)  
 353 is also in  $s$ , thus it is mapped to make the diagram of Fig. 4 commutative. The node,  
 354 say  $q$ , in class (1) is mapped from a node in  $t$ , that is mapped to the root of the redex  
 355 in  $s$ . Let  $p$  the root of the contractum of this redex. Thus, map  $q$  to  $p$ . This define a  
 356 homomorphism which is adequate.  $\square$

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

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

365 *Proof.* By induction on the length of a derivation.  $\square$

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

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

## 379 6 Performance

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

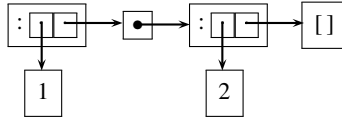


Fig. 4: The evaluation of `append [1] [2]` produces a list containing a forward node represented by the large black dot in the above diagram.

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

394 The execution time of each program is too short to be reliably measured with ordi-  
 395 nary tools. As far as memory consumption is concerned, our technique saves the al-  
 396 location and the traversal of the forward node. There is a similar program that instead

<sup>1</sup> The word “collapse” is overloaded in graph parlance. In this context, its refers to a relation on graphs defined in the cited reference.

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

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

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

## 419 7 Narrowing

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

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

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

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

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

## 455   8   Discussion and Related Work

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

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

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

## 477 9 Acknowledgments

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

## 481 References

- 482 1. M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs.  
483 *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
- 484 2. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third*  
485 *International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra,  
486 Italy, September 1992. Springer LNCS 632.
- 487 3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the*  
488 *Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–  
489 30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at <http://cs.pdx.edu/~antoy/homepage/publications/alp97/full.pdf>.  
490
- 491 4. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic*  
492 *Computation*, 40(1):875–903, 2005.
- 493 5. S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522,  
494 May 2010.
- 495 6. S. Antoy and M. Hanus. Functional logic programming. *Comm. of the ACM*, 53(4):74–85,  
496 April 2010.
- 497 7. S. Antoy and A. Jost. Are needed redexes really needed? In *Proceedings of the 15th Symposi-*  
498 *um on Principles and Practice of Declarative Programming*, PPDP '13, pages 61–71, New  
499 York, NY, USA, 2013. ACM.
- 500 8. S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of*  
501 *the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31,  
502 Kobe, Japan, May 2012. Springer LNCS 7294.
- 503 9. Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. *FUNDAMENTA*  
504 *INFORMATICA*, 26, 1996.
- 505 10. Zena M. Ariola, Jan Willem Klop, and Detlef Plump. Bisimilarity in term graph rewriting.  
506 *Information and Computation*, 156(12):2 – 24, 2000.
- 507 11. H. P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and  
508 M.R. Sleep. Term graph rewriting. In *PARLE Parallel Architectures and Languages Europe*,  
509 volume LNCS 259, pages 141–158, Eindhoven, 1987. Springer-Verlag. also technical report  
510 SYS-C87-01.
- 511 12. M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge Univer-  
512 sity Press, 2003.
- 513 13. T. J.W. Clarke, P. J.S. Gladstone, C. D. MacLean, and A. C. Norman. Skim - the s, k, i  
514 reduction machine. In *Proceedings of the 1980 ACM Conference on LISP and Functional*  
515 *Programming*, LFP '80, pages 128–135, New York, NY, USA, 1980. ACM.
- 516 14. R. Echahed. Inductively sequential term-graph rewrite systems. In *Graph Transformations,*  
517 *4th International Conference (ICGT 2008)*, pages 84–98, Leicester, UK, 2008. Springer,  
518 LNCS 5214.
- 519 15. R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Techni-  
520 cal Report 985-I, IMAG, 1997. Available at [ftp://ftp.imag.fr/pub/labo-LEIBNIZ/](ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz)  
521 [OLD-archives/PMP/c-graph-rewriting.ps.gz](ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz).

- 522 16. W. Fokkink and J. van de Pol. Simulation as a correct transformation of rewrite systems.  
523 In *In Proceedings of 22nd Symposium on Mathematical Foundations of Computer Science,*  
524 *LNCS 1295*, pages 249–258. Springer, 1997.
- 525 17. J. R. W. Glauert, R. Kennaway, G. A. Papadopoulos, and M. R. Sleep. Dactl: an experimental  
526 graph rewriting language. *J. Prog. Lang.*, 5(1):85–108, 1997.
- 527 18. M. Hanus. The integration of functions into logic programming: From theory to practice.  
528 *Journal of Logic Programming*, 19&20:583–628, 1994.
- 529 19. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics -*  
530 *Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
- 531 20. M. Hanus, editor. *PAKCS 1.11.4: The Portland Aachen Kiel Curry System*, 2014. Available  
532 at <http://www.informatik.uni-kiel.de/~pakcs>.
- 533 21. G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems, I. In J.-L.  
534 Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*,  
535 pages 395–414. MIT Press, Cambridge, MA, 1991.
- 536 22. J. F. T. Kamperman and H. R. Walters. Simulating TRSs by minimal TRSs a simple, efficient,  
537 and correct compilation technique. Technical Report CS-R9605, CWI, 1996.
- 538 23. J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. de Vries. The adequacy of term graph  
539 rewriting for simulating term rewriting. In M. R. Sleep, M. J. Plasmeijer, and M. C. J. D.  
540 van Eekelen, editors, *Term Graph Rewriting Theory and Practice*, pages 157–169. J. Wiley  
541 & Sons, Chichester, UK, 1993.
- 542 24. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320,  
543 January 1964.
- 544 25. D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg,  
545 editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.
- 546 26. Detlef Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51:277–  
547 289, 2001.
- 548 27. M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term Graph Rewriting*  
549 *Theory and Practice*. J. Wiley & Sons, Chichester, UK, 1993.