Lifting Term Rewriting Derivations in Constructor Systems by Using Generators*

Adrián Riesco
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Juan Rodríguez-Hortalá
Lambdoop Solutions
juan.rodriguez@lambdoop.com

Narrowing is a procedure that was first studied in the context of equational E-unification and that has been used in a wide range of applications. The classic completeness result due to Hullot states that any term rewriting derivation starting from an instance of an expression can be 'lifted' to a narrowing derivation, whenever the substitution employed is normalized. In this paper we adapt the generator-based extra-variables-elimination transformation used in functional-logic programming to overcome that limitation, so we are able to lift term rewriting derivations starting from arbitrary instances of expressions. The proposed technique is limited to left-linear constructor systems and to derivations reaching a ground expression. We also present a Maude-based implementation of the technique, using natural rewriting for the on-demand evaluation strategy.

1 Introduction

Narrowing [2] is a procedure that was first studied in the context of equational E-unification and that has been used in a wide range of applications [18, 20]. Narrowing can be described as a modification of term rewriting in which matching is replaced by unification so, in a derivation starting from a goal expression, it is able to deduce the instantiation of the variables of the goal expression that is needed for the computation to progress. The key result for the completeness of narrowing w.r.t. term rewriting is *Hullot's lifting lemma* [12], which states that any term rewriting derivation $e_1\theta \to^* e_2$ can be *lifted* into a narrowing derivation $e_1 \leadsto^*_\sigma e_3$ such that e_3 and σ are more general than e_2 and θ—w.r.t. to the usual instantiation preorder [3], and for the variables involved in the derivations—, provided that the starting substitution θ is normalized [19]. This latter condition is essential, so it is fairly easy to break Hullot's lifting lemma by dropping it: e.g. under the term rewriting system (TRS) $\{f(0,1) \to 2, coin \to 0, coin \to 1\}$ the term rewriting derivation $f(X,X)[X/coin] \to^* 2$ cannot be lifted by any narrowing derivation. Several variants and extensions of narrowing have been developed in order to improve that result under certain assumptions or for particular classes of term rewriting systems [19, 18, 9].

In this paper we show how to adapt the generator-based extra variable elimination transformation used in functional-logic programming (FLP) to drop the normalization condition required by Hullot's lifting lemma. The proposed technique is devised for left-linear constructor systems (CS's) with extra variables, and it is limited to derivations reaching a ground expression. To test the feasibility of this approach, we have also developed a prototype in Maude [6], relying on the natural rewriting on-demand strategy [10] to obtain an effective operational procedure.

The rest of the paper is organized as follows. In Section 2 we introduce the semantics for CS's that we have used to formally prove the results, and that first suggested us the feasibility of the approach. In Section 3 we show our adaptation of the generators technique from FLP, and use the semantics for

*Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04).

S. Escobar (Ed.): XIV Jornadas sobre Programación
y Lenguajes, PROLE 2014, Revised Selected Papers
EPTCS 173, 2015, pp. 87–99, doi:10.4204/EPTCS.173.7

proving the adequacy of the technique for lifting term rewriting derivations reaching ground c-terms. In Section 4 we outline the implementation and commands of our prototype. Finally Section 5 concludes and outlines some lines of future work.

2 Preliminaries and formal setting

We mostly use the notation from [2], with some additions from [15]. We consider a first order signature $\Sigma = CS \uplus FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols respectively, all of them with associated arity. We use $c,d,\ldots$ for constructors, $f,g,\ldots$ for functions and $X,Y,\ldots$ for variables of a numerable set $\mathscr{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. The set Exp of *total expressions* is defined as $Exp \ni e ::= X \mid h(e_1,\ldots,e_n)$, where $X \in \mathscr{V}$, $h \in CS^n \cup FS^n$ and $e_1,\ldots,e_n \in Exp$. The set $CTerm$ of *total constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e,e',\ldots$ for expressions and $t,s,\ldots$ for c-terms. We say that an expression e is *ground* iff no variable appears in e. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathscr{C} ::= [\,] \mid h(e_1,\ldots,\mathscr{C},\ldots,e_n)$.

Example 1 *We will use a simple example throughout this section to illustrate these definitions. Assume we want to represent the staff of a shop, so we have* $CS = \{madrid^0, vigo^0, man^0, woman^0, pepe^0, luis^0, pilar^0, maria^0, e^2, p^2\}$*, where e will be the constructor for employees and p the constructor for pairs, and* $FS = \{branches^0, search^1, employees^1\}$*. Using this signature, we can build the set* $Exp = \{madrid, vigo, employees\ (madrid), p(pilar,X),\ldots\}$*. From this set, we have* $CTerm = \{madrid, vigo, p(pilar,X),\ldots\}$*, while the ground terms are* $\{employees(madrid), madrid, vigo,\ldots\}$*. Finally, a possible one-hole context is* $p([\,],X)$*.*

We also consider the extended signature $\Sigma_\bot = \Sigma \cup \{\bot\}$, where $\bot$ is a new 0-arity constructor symbol that does not appear in programs and which stands for the undefined value. Over this signature we define the sets $Exp_\bot$ and $CTerm_\bot$ of *partial* expressions and c-terms, respectively. The intended meaning is that $Exp_\bot$ and $CTerm_\bot$ stand for partial expressions and values, respectively. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathscr{C}[e] \sqsubseteq \mathscr{C}[e']$ for all $e,e' \in Exp_\bot, \mathscr{C} \in Cntxt$. The *shell* $|e|$ of an expression e represents the outer constructed part of e and is defined as: $|X| = X$; $|c(e_1,\ldots,e_n)| = c(|e_1|,\ldots,|e_n|)$; $|f(e_1,\ldots,e_n)| = \bot$. It is trivial to check that for any expression e we have $|e| \in CTerm_\bot$, that any total expression is maximal w.r.t. $\sqsubseteq$, and that as consequence if t is total then $t \sqsubseteq |e|$ implies $t = e$.

Example 2 *Using the signature from Example 1, we have* $employees(\bot) \in Exp_\bot$*,* $p(\bot,X) \in CTerm_\bot$*, and* $|p(search(branches),X)| = p(\bot,X)$*.*

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathscr{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ε for the identity (or empty) substitution. We write $e\theta$ to apply of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and variable range of θ are defined as $dom(\theta) = \{X \in \mathscr{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. By $[X_1/e_1,\ldots,X_n/e_n]$ we denote a substitution σ such that $dom(\sigma) = \{X_1,\ldots,X_n\}$ and $\forall i.\sigma(X_i) = e_i$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given $W \subseteq \mathscr{V}$ we write $\theta|_W$ for the restriction of θ to W, i.e. $(\theta|_W)(X) = \theta(X)$ if $X \in W$, and $(\theta|_W)(X) = X$ otherwise; we use $\theta|_{\setminus D}$ as a shortcut for $\theta|_{(\mathscr{V} \setminus D)}$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for

all $X \in dom(\theta)$. We say a substitution σ is ground iff $vran(\sigma) = \emptyset$, i.e. $\forall X \in dom(\sigma)$ we have that $\sigma(X)$ is ground. The sets $Subst_\perp$ and $CSubst_\perp$ of partial substitutions and partial c-substitutions are the sets of finite mappings from variables to partial expressions and partial c-terms, respectively.

Example 3 *Using the signature from Example 1, we can define the C-substitutions* $\theta_1 \equiv X/woman$, $\theta_2 \equiv X/man$, *and* $\theta_3 \equiv Y/pilar$. *We can define the restrictions* $\theta_1|_{\{X\}} = \theta_1$ *and* $\theta_1|_{\setminus\{X\}} = \varepsilon$. *Finally, given the expression* $p(X,Y)$ *we have* $p(X,Y)\theta_1\theta_2 = p(woman,Y)$ *and* $p(X,Y)\theta_1\theta_3 = p(X,Y)\theta_3\theta_1 = p(woman,pilar)$.

A left-linear constructor-based term rewriting system or just *constructor system* (*CS*) or *program* $\mathscr{P}$ is a set of c-rewrite rules of the form $f(\bar{t}) \to r$ where $f \in FS^n$, $r \in Exp$ and $\bar{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\bar{t}$. Notice that we allow r to contain so called *extra variables*, i.e., variables not occurring in $f(\bar{t})$. To be precise, we say that $X \in \mathscr{V}$ is an extra variable in the rule $l \to r$ iff $X \in var(r) \setminus var(l)$, and by $vExtra(R)$ we denote the set of extra variables in a program rule R. We assume that every CS contains the rules $\mathscr{Q} = \{X \ ? \ Y \to X, X \ ? \ Y \to Y\}$, defining the behavior of $? \in FS^2$, used in infix mode, and that those are the only rules for ?. Besides, ? is right-associative so $e_1 \ ? \ e_2 \ ? \ e_3$ is equivalent to $e_1 \ ? \ (e_2 \ ? \ e_3)$. For the sake of conciseness we will often omit these rules when presenting a CS. A consequence of this is that we only consider non-confluent programs. Given a TRS $\mathscr{P}$, its associated *term rewriting relation* $\to_\mathscr{P}$ is defined as: $\mathscr{C}[l\sigma] \to_\mathscr{P} \mathscr{C}[r\sigma]$ for any context $\mathscr{C}$, rule $l \to r \in \mathscr{P}$ and $\sigma \in Subst$. We write $\overset{*}{\to}_\mathscr{P}$ for the reflexive and transitive closure of the relation $\to_\mathscr{P}$. We will usually omit the reference to $\mathscr{P}$ or denote it by $\mathscr{P} \vdash e \to e'$ and $\mathscr{P} \vdash e \to^* e'$.

Example 4 *Using the signature from Example 1, we can describe the following program:*

$$\begin{array}{lcl}
branches & \to & madrid \ ? \ vigo \\
employees(madrid) & \to & e(pepe,men) \\
employees(madrid) & \to & e(maria,men) \\
employees(vigo) & \to & e(pilar,women) \ ? \ e(luis,men) \\
search(e(N,S)) & \to & p(N,N)
\end{array}$$

In this example, the function symbol branches defines the different branches of the company, employees defines the employees in each branch (built with the constructor symbol e), and search returns a pair of names, built with the constructor symbol p. Note that several different notations are possible; for example, it is possible to define the employees of one branch by using just one rule and the ? operator or just several different rules with the same lefthand side.

2.1 A proof calculus for constructor systems with extra variables

In [15] an adequate semantics for reachability of c-terms by term rewriting in CS's was presented. The key idea there was using a suitable notion of value, in this case the notion of s-cterm. *SCTerm* is the set of s-cterms, which are *finite* sets of elemental s-cterms, while the set *ESCTerm* of elemental s-cterms is defined as $ESCTerm \ni est ::= X \mid c(st_1,\ldots,st_n)$ for $X \in \mathscr{V}$, $c \in CS^n$, $st_1,\ldots,st_n \in SCTerm$. We extend this idea to expressions obtaining the sets *SExp* of s-expressions or just s-exp, and *ESExp* of elemental s-expressions, which are defined the same but now using any symbol in Σ in applications instead of just constructor symbols. Note that the s-expression $\emptyset$ corresponds to $\perp$, so s-exps are partial by default. The approximation preorder $\sqsubseteq$ is defined for s-exps as the least preorder such that $se \sqsubseteq se'$ iff $\forall ese \in se.\exists ese' \in se'$ such that $ese \sqsubseteq ese'$, $X \sqsubseteq X$ for any $X \in \mathscr{V}$, and $h(se_1,\ldots,se_n) \sqsubseteq h(se'_1,\ldots,se'_n)$ iff $\forall i.se_i \sqsubseteq se'_i$.

E	$se \twoheadrightarrow \emptyset$	
RR	$\{X\} \twoheadrightarrow \{X\}$	if $X \in \mathscr{V}$
DC	$\dfrac{se_1 \twoheadrightarrow st_1 \ \ldots \ se_n \twoheadrightarrow st_n}{\{c(se_1,\ldots,se_n)\} \twoheadrightarrow \{c(st_1,\ldots,st_n)\}}$	if $c \in CS$
MORE	$\dfrac{se \twoheadrightarrow st_1 \ \ldots se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \ldots \cup st_n}$	
LESS	$\dfrac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{\{ese_1,\ldots,ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m}$	if $n \geq 2, m > 0$, for any $\{esa_1,\ldots,esa_m\} \subseteq \{ese_1,\ldots,ese_n\}$
ROR	$\dfrac{se_1 \twoheadrightarrow \widetilde{p_1}\theta \ \ldots \ se_n \twoheadrightarrow \widetilde{p_n}\theta \ \ \widetilde{r}\theta \twoheadrightarrow st}{\{f(se_1,\ldots,se_n)\} \twoheadrightarrow st}$	if $(f(p_1,\ldots,p_n) \to r) \in \mathscr{P}$, $\theta \in SCSubst$

Figure 1: A proof calculus for constructor systems

Example 5 *Using the signature from Example 1, and given the s-cterm* $sct \equiv e(\{pepe, pilar\}, \{men, women\})$*, we have* $sct \in ESCTerm$*, while* $\{sct\} \in SCTerm$*. Similarly, given the es-exp* $esex \equiv employees(\{madrid, vigo\})$ *we have* $esex \in ESExp$ *and* $esex \notin ESCTerm$*. Finally, we have that* $\{esex\} \in SExp$*.*

The sets *SSubst* and *SCSubst* of s-substitutions and s-csubstitutions (or just s-csubst) consist of finite mappings from variables to s-exps or s-cterms, respectively. We extend s-substs to be applied to *ESExp* and *SExp* as $\sigma : ESExp \to SExp$ defined by $X\sigma = \sigma(X)$, $h(\overline{se})\sigma = \{h(\overline{se\sigma})\}$; and $\sigma : SExp \to SExp$ defined by $se\sigma = \bigcup_{ese \in se} ese\sigma$. The approximation preorder $\sqsubseteq$ is defined for s-substs as $\sigma \sqsubseteq \theta$ iff $\forall X \in \mathscr{V}.\sigma(X) \sqsubseteq \theta(X)$. For any nonempty and finite set $\{\theta_1,\ldots,\theta_n\} \subseteq SCSubst$ we define $\bigcup\{\theta_1,\ldots,\theta_n\} \in SCSubst$ as $\bigcup\{\theta_1,\ldots,\theta_n\}(X) = \theta_1(X) \cup \ldots \cup \theta_n(X)$.

Example 6 *Using the signature from Example 1, we can define the s-csubstitution* $\sigma \equiv \{X/\{pepe, pilar\}, Y/\{men, women\}\} \in SCSubst$*. Hence, given* $esex \equiv e(\{X\},\{Y\}) \in ESExp$ *we have that* $esex\sigma \equiv e(\{pepe, pilar\}, \{men, women\})$*.*

We obtain the denotation of an expression as the denotation of its associated s-expression, assigned by the operator $\widetilde{\ }: Exp_\perp \to SExp$, defined as $\widetilde{\perp} = \emptyset$; $\widetilde{X} = \{X\}$ for any $X \in \mathscr{V}$; $\widetilde{h(e_1,\ldots,e_n)} = \{h(\widetilde{e_1},\ldots,\widetilde{e_n})\}$ for any $h \in \Sigma^n$. The operator $\widetilde{\ }$ is extended to s-substitutions as $\widetilde{\sigma}(X) = \widetilde{\sigma(X)}$, for $\sigma \in Subst_\perp$. It is easy to check that $\widetilde{e\sigma} = \widetilde{e}\widetilde{\sigma}$ (see [15]). Conversely, we can flatten an s-expression *se* to obtain the set $flat(se)$ of expressions "contained" in it, so $flat(\emptyset) = \{\perp\}$ and $flat(se) = \bigcup_{ese \in se} flat(ese)$ if $se \neq \emptyset$, where the flattening of elemental s-exps is defined as $flat(X) = \{X\}$; $flat(h(se_1,\ldots,se_n)) = \{h(e_1,\ldots,e_n) \mid e_i \in flat(se_i) \text{ for } i = 1..n\}$.

Example 7 *Using the signature from Example 1, we have that* $\widetilde{p(X,Y)} = \{p(\{X\},\{Y\})\}$ *and* $flat(\{p(\{X\}, \{Y,Z\})\}) = \{p(X,Y), p(X,Z)\}$

In Figure 1 we can find the proof calculus that defines the semantics of s-expressions. Our proof calculus proves reduction statements of the form $se \twoheadrightarrow st$ with $se \in SExp$ and $st \in SCTerm$, expressing that *st* represents an approximation to one of the possible structured sets of values for *se*. We refer the interested reader to [15] for detailed explanations about the calculus.We write $\mathscr{P} \vdash se \twoheadrightarrow st$ to express that $se \twoheadrightarrow st$ is derivable in our calculus under the CS $\mathscr{P}$. We say that a proof for a statement $\mathscr{P} \vdash se \twoheadrightarrow st$ is ground iff *se*, *st* and all the s-exp in the premises are ground. The *denotation* of an s-expression *se* under

a CS $\mathscr{P}$ is defined as $[\![se]\!]^{\mathscr{P}} = \{st \in SCTerm \mid \mathscr{P} \vdash se \twoheadrightarrow st\}$, so $[\![e]\!]^{\mathscr{P}} = [\![\widetilde{e}]\!]^{\mathscr{P}}$. In the following we will usually omit the reference to $\mathscr{P}$. The denotation of $\sigma \in SSubst$ is defined as $[\![\sigma]\!] = \{\theta \in SCSubst \mid \forall X \in \mathscr{V}, \sigma(X) \twoheadrightarrow \theta(X)\}$, so for $\theta \in Subst_\bot$ we define $[\![\theta]\!] = [\![\widetilde{\theta}]\!]$.

Example 8 *Using the signature from Example 1 and the rules from Example 4, we have* $employees(\{X\}) \twoheadrightarrow \{e(pepe,men)\}$*, given the substitution* $X/\{madrid\}$*. Moreover, we can use this same substitution to reach* $\{e(maria,men)\}$ *by using a different program rule.*

The setting presented in [15] was not able to deal with extra variables. As programs with extra variables are very common when using narrowing, for this work we decided to extend the setting to deal with them. But then we realized that the semantics had the foundations to deal with extra variables, as the rule **ROR** from Figure 1 allows to instantiate extra variables freely with s-cterms: therefore all that was left was proving the adecuacy of the semantics in this extended scenario. Nevertheless, as a consequence of the freely instantiation of extra variables in **ROR**, then every program with extra variables turns into non-deterministic. For example consider a program $\{f \to (X,X)\}$ for which the constructors $0,1 \in CS^0$ are available, then we can do:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{0\} \twoheadrightarrow \{0\}}\,\textbf{DC}}{\{0,1\} \twoheadrightarrow \{0\}}\,\textbf{LESS} \quad \dfrac{\ldots}{\{0,1\} \twoheadrightarrow \{1\}}}{\widetilde{(X,X)}[X/\{0,1\}] = \{(\{0,1\},\{0,1\})\} \twoheadrightarrow \{(\{0\},\{1\})\}}\,\textbf{DC}}{\widetilde{f} = \{f\} \twoheadrightarrow \{(\{0\},\{1\})\} = \widetilde{(0,1)}}\,\textbf{ROR}$$

But in fact this is not very surprising, and it has to do with the relation between non-determinism and extra variables [1], but adapted to the run-time choice semantics [13, 23] induced by term rewriting. As a consequence of this we assume that all the programs contain the function ?, so we only consider non-confluent TRS's. We admit that this is a limitation of our setting, but we also conjecture that for confluent TRS's a simpler semantics could be used, for which the packing of alternatives of c-terms would not be needed. However, the important point to bear in mind is that having ? at one's disposal is enough to express the non-determinism of any program [11], so we can use it to define the transformation $\widehat{\ }$ from s-exp and elemental s-exp to partial expressions that, contrary to *flat*, now takes care of the keeping the nested set structure by means of uses of the ? function. Then $\widehat{\ }: ESExp \to Exp_\bot$ is defined by $\widehat{X} = X$, $\widehat{h(se_1,\ldots,se_n)} = h(\widehat{se_1},\ldots,\widehat{se_n})$; and $\widehat{\ }: SExp \to Exp_\bot$ is defined by $\widehat{\emptyset} = \bot$, $\widehat{\{ese_1,\ldots,ese_n\}} = \widehat{ese_1}$? … ? $\widehat{ese_n}$ for $n > 0$, where in the case for $\widehat{\{ese_1,\ldots,ese_n\}}$ we use some fixed arbitrary order on terms in the line of Prolog [24] for arranging the arguments of ?. This operator is also overloaded for substitutions as $\widehat{\ }: SSubst \to Subst_\bot$ as $(\widehat{\sigma})(X) = \widehat{\sigma(X)}$. Thanks to the power of ? to express non-determinism, that transformation preserves the semantics from Figure 1, and we can use it to prove the following new result about the adequacy of the semantics for programs with extra variables—see [21] for a detailed proof.

Theorem 1 (Adequacy of $[\![\_]\!]$) *For all* $e,e' \in Exp, t \in CTerm_\bot, st \in SCTerm$*:*
Soundness *$st \in [\![\widetilde{e}]\!]$ and $t \in flat(st)$ implies $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Therefore, $\widetilde{t} \in [\![\widetilde{e}]\!]$ implies $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Besides, in any of the previous cases, if t is total then $e \to^* t$.*
Completeness *$e \to^* e'$ implies $\widetilde{|e'|} \in [\![\widetilde{e}]\!]$. Hence, if t is total then $e \to^* t$ implies $\widetilde{t} \in [\![\widetilde{e}]\!]$.*

We refer the interested reader to [15] and [14] (Theorems 2 and 3) for more properties of $[\![\_]\!]$ like compositionality or monotonicity, some of which are used in the proofs for the results in the present paper.

$\_ \vdash \_ \lessdot \_ \subseteq CS \times SCTerm \times Exp$	$\_ \vdash \_ \lessdot \_ \subseteq CS \times ESCTerm \times Exp$
$\mathscr{P} \vdash st \lessdot e$ if $\forall est \in st, \mathscr{P} \vdash est \lessdot e$	$\mathscr{P} \vdash X \lessdot e$ if $\mathscr{P} \vdash e \to^* X$
	$\mathscr{P} \vdash c(\overline{st}) \lessdot e$ if $\mathscr{P} \vdash e \to^* c(\overline{e})$ for some $\overline{e}$ such that $\forall e_i \in \overline{e}, \mathscr{P} \vdash st_i \lessdot e_i$

Figure 2: Domination relation

There is another characterization of $[\![\_]\!]$ closer to term rewriting which is based of the *domination relation* $\_ \lessdot \_$ presented in Figure 2 (we will omit the prefix "$\mathscr{P} \vdash$" when it is implied by the context). With this relation we try to transfer to the rewriting world the finer distinction between sets of values that the structured representation of *SCTerm* allows us to perform. We extend the relation $\_ \lessdot \_$ to $\_ \vdash \_ \lessdot \_ \subseteq CS \times SCSubst \times Subst$ by $\theta \lessdot \sigma$ iff $\forall X \in \mathscr{V}, \theta(X) \lessdot \sigma(X)$. As can be seen in [14], this relation is a key ingredient to prove the soundness of $[\![\_]\!]$, and its equivalence to $[\![\_]\!]$ is stated in the following result.

Lemma 1 (Domination) *For all $e \in Exp, st \in SCTerm$, $st \in [\![\widetilde{e}]\!]$ iff $st \lessdot e$. Besides, regarding substitutions, for all $\sigma \in Subst$, $\theta \in SCSubst$ we have that $\theta \in [\![\widetilde{\sigma}]\!]$ iff $\theta \lessdot \sigma$.*

3 The generators approach

In this section we will show a proposal for adapting the generators technique from the field of functional-logic programming [8, 1] to the lifting of term rewriting derivations from arbitrary instances of expressions. This technique consists in replacing free and extra variables by a call to a *generator function* that can be reduced to any ground c-term. The generator function *gen* is defined as follows:

Definition 1 (Generator function) *For any program $\mathscr{P}$ we can define a fresh function gen as follows: for each $c \in CS^n$ we add a new rule $gen \to c(gen, \ldots, gen)$ to the program. By $\mathscr{G}$ we denote the program that consists of the set of rules for gen.*

Example 9 *Given the system in Example 4, the rules for gen are $\mathscr{G} \equiv \{gen \to madrid, gen \to vigo, gen \to pepe, gen \to luis, gen \to maria, gen \to pilar, gen \to men, gen \to women,\ gen \to e(gen,\ gen),\ gen \to p(gen, gen)\}$.*

The point with *gen* is that we can use it to compute any *ground value*:

Proposition 1 *For all $t \in CTerm$, $st \in SCTerm$ and $\theta \in SCSubst$ such that those are ground we have $gen \to^* t$, $st \in [\![gen]\!]$ and $\theta \in [\![[\overline{X/gen}]]\!]$ for $\overline{X} = dom(\theta)$.*

Then the main idea with generators is that given some $e \in Exp$ with $var(e) = \overline{X}$, we can simulate narrowing with e by performing term rewriting with $e[\overline{X/gen}]$. As *gen* can be reduced to any ground s-cterm, then Lemma 1 from [15] suggests that this procedure will be able to lift derivations $e\sigma \to^* t$ with an arbitrary $\sigma \in Subst$, even those which are not normalized: e.g. we can easily apply this technique to the example in Section 1, getting $f(X,X)[X/gen] \to^* f(0,1) \to 2$. Sadly, on the other hand, only derivations reaching a ground c-term will be lifted, and the reason for that is that *gen* can be reduced to an arbitrary ground c-term, but it cannot be reduced to any c-term with variables. Thus, under the program $\{g(c(X)) \to X\}$ the term rewriting derivation $g(Y)[Y/c(X)] \to X$ cannot be lifted by using generators, as $g(Y)[Y/gen] \to g(c(gen)) \to gen \not\to^* X$, even though $[Y/c(X)]$ is a normalized substitution.

In order to prove the completeness of the generators technique for the reachability of ground c-terms, we rely on the following modification of Lemma 1 from [15].

Lemma 2 *For all $\sigma \in SSubst$, $se \in SExp$, $st \in SCTerm$, if st is ground then $se\sigma \twoheadrightarrow st$ implies $\exists \theta \in [\![\sigma]\!]$ such that $se\theta \twoheadrightarrow st$, θ is ground and $dom(\theta) = dom(\sigma)$.*

Note the restriction to ground s-cterms in Lemma 2 is crucial, and that it reflects the lack of completeness for reaching non-ground c-terms of the generators technique: e.g. under the program $\{f \to c(X)\}$ using $se = \{Y\}$, $\sigma = [Y/\{f\}]$ and $st = \{c(\{X\})\}$ the only $\theta \in [\![[Y/\{f\}]]\!]$ fulfilling the first condition is $\theta = [Y/\{c(\{X\})\}]$, which is not ground. On the other hand those s-csubst obtained by Lemma 2 are ground, and so they are in the denotation of an appropriate substitution with only generators in its range.

Generators can be introduced in programs systematically in order to eliminate extra variables from program rules using a program transformation in the line of those from [8, 1]. In those works the usual call-time choice semantics for functional-logic programming [20] was adopted, therefore we use a different transformation that is adapted to the use of term rewriting, which leads to a different set of reachable c-terms than that obtained with call-time choice [23]. The point in eliminating extra variables is that in this way we eliminate the "oracular guessing" that is performing in a term rewriting step using extra variables: by this guessing we refer for example to the instantiation performed under the program $\{f \to g(X), g(0) \to 1\}$ in the first step of the derivation $f \to g(0) \to 1$ for the extra variable X, that has to be instantiated with 0 in order for the derivation to continue. That, combined with a suitable on-demand evaluation strategy like natural rewriting [10], turns term rewriting with generators into an effective mechanism for lifting term rewriting derivations. We formalize our extra variable elimination transformation through the following definition.

Definition 2 (Generators program transformation)
Given a program $\mathscr{P}$ its transformation $\hat{\mathscr{P}}$ consists of the rules $\mathscr{G}$ for gen together with the transformation of each rule in $\mathscr{P}$, defined as $\widehat{f(p_1,\ldots,p_n)} \to r = f(p_1,\ldots,p_n) \to r[\overline{X/gen}]$, where $\overline{X} = vExtra(f(p_1,\ldots,p_n) \to r)$.

Then it is clear that for any program $\mathscr{P}$ its transformation $\hat{\mathscr{P}}$ does not have any extra variable in its rules. Note that, contrary to the proposals from [8, 1], this transformation destroys the sharing that normally appears when there are several occurrences of the same variable, in procedures that instantiate variables like narrowing or SLD resolution. In our transformation, however, once instantiated with *gen* every occurrence of the same variable evolves independently. This is needed to ensure completeness under the transformed program, which can be seen considering the program $\mathscr{P} = \{f \to (g(X), h(X), g(0) \to 1, h(1) \to 2\}$ and the derivation $\mathscr{P} \vdash f \to (g(0\,?\,1), h(0\,?\,1)) \to^* (g(0), h(1)) \to^* (1,2)$: as extra variables can be instantiated with arbitrary expressions that implies that in particular those can be instantiated with "alternatives" of expressions built using the ? function, which can evolve independently after the alternative between them is resolved. We can lift that derivation with our transformation as $\hat{\mathscr{P}} \vdash f \to (g(gen), h(gen)) \to^* (g(0), h(1)) \to^* (1,2)$. The adequacy of the transformation is formulated in the following result, in the same terms as the variable elimination result from [8].

Theorem 2 *For any program $\mathscr{P}$, $se \in SExp$, $st \in SCTerm$ if st is ground then $\mathscr{G} \uplus \mathscr{P} \vdash se \twoheadrightarrow st$ iff $\hat{\mathscr{P}} \vdash se \twoheadrightarrow st$.*

After eliminating extra variables with the proposed program transformation, we can then emulate the instantiation of variables performed by a narrowing procedure by just replacing free variables with *gen*, thus lifting any term rewriting derivation starting from an arbitrary instance of an expression to a ground c-term.

Theorem 3 (Lifting) *For any program $\mathscr{P}$, $e, e' \in Exp$ such that e' is ground:*
Soundness *$\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* e'$ implies $\exists \sigma \in Subst$ such that $\mathscr{P} \vdash e\sigma \to^* e''$ for some $e'' \in Exp$ such that $|e'| \sqsubseteq |e''|$ with $dom(\sigma) = \overline{X}$. As a consequence, if $e' = t \in CTerm$ then $\mathscr{P} \vdash e\sigma \to^* t$.*
Completeness *For any $\sigma \in Subst$ we have that $\mathscr{P} \vdash e\sigma \to^* e'$ implies $\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* e''$ for some $e'' \in Exp$ such that $|e'| \sqsubseteq |e''|$ with $\overline{X} = dom(\sigma)$. As a consequence, if $e' = t \in CTerm$ then $\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* t$.*

4 Maude prototype

We present in this section our prototype; much more information can be found at `http://gpd.sip.ucm.es/snarrowing`. The prototype is started by typing `loop init-s .`, that initiates an input/output loop where programs and commands can be introduced. These programs have syntax `smod NAME is STMNTS ends`, where `NAME` is the identifier of the program and `STMNTS` is a sequence of constructor-based left-linear rewrite rules. For instance, Example 4 would be written as follows:

```
(smod CLERKS is
  branches -> madrid ? vigo .
  employees(madrid) -> e(pepe, men) .
  employees(madrid) -> e(maria, men) .
  employees(vigo) -> e(pilar, women) ? e(luis, men) .
  search(e(N,S)) -> p(N,N) .
ends)
```

where upper-case letters are assumed to be variables. We can evaluate terms with variables with the command `eval-gen`, that transforms each variable in the term into the `gen` constant described above and evaluates the thus obtained expression in the module extended with the `gen` rules:

```
Maude> (eval-gen search(X,X) .)
Result: p(madrid, madrid)
```

That is, the tool first finds a result with the same value for the two elements of the pair. We can ask the system for more solutions with the `next` command until no more solutions are found, which will reveal pairs with different values:

```
Maude> (next .)
Result: p(madrid,vigo)
```

Finally, the system combines the on-demand strategy with two different search strategies: depth-first and breadth-first, and allows the user to check the trace in order to see how the generators are instantiated. We will show in the following section how to use these commands.

4.1 Looking for alternatives

We present here a more complex example, which introduces how to use our tool to search for different paths leading to the solution. This example presents a simplified version of the intruder protocol introduced in [22], which is also executable with the generators approach presented here and is available at `http://gpd.sip.ucm.es/snarrowing`.

The module `PARTY` below describes the specification of a party. Our goal in this party is to have fun, so we define the function `success`, which receives a set of friends `F` and a set of elements that we already have. It is reduced to the function `haveFun` applied to the set obtained after calling to our friends:

```
(smod PARTY is

  success(F, S) -> haveFun?(makeCalls(F, S)) .
```

The function `haveFun` is reduced to `tt` (standing for the value `true`) when it receives the constant `fun`:

```
haveFun(fun) -> tt .
```

The function `makeCalls` combines the current items with the ones obtained by making further calls using the new items obtained by offering your items to your friends:

```
  makeCalls(F, S) -> S ? makeCalls(F, makeAnOffer(F, S)) .
```

We can reach different results by using `makeAnOffer`. First, it is possible to combine the current items to obtain a new one:

```
  makeAnOffer(F, S) -> combine(S, S) .
```

This combination, achieved by the `combine` function, generates a `burger` from `bread` and `meat`, and `fun` when a `burger` and a `videogame` are found:

```
  combine(bread, meat) -> burger .
  combine(burger, videogames) -> fun .
```

Another possibility is to call a friend and show him the items we have obtained so far:

```
  makeAnOffer(F, S) -> call(F, S) .
```

This `call` depends on the friend we call. We present below the different possibilities:

```
  call(enrique, drink) -> music .
  call(adri, meat) -> bread .
  call(rober, music) -> videogames .
  call(nacho, videogames) -> music .
  call(juan, food) -> drink .
ends)
```

Once this module is loaded into the interpreter, we indicate that we want to activate the path. In this way, we can explore the different ways to reach the values:

```
Maude> (path on.)
Path activated.
```

We also set the exploration strategy to *breadth first*, so the tool finds the shortest solutions first:

```
Maude> (breadth-first .)
Breadth-first strategy selected.
```

We can now look for solutions to the `success` function, using a variable as argument:

```
Maude> (eval-gen success(F, S) .)
Result: tt
```

We can now examine the path traversed by the tool to reach the result as follows:

```
Maude> (show path .)
haveFun(makeCalls(gen,gen))
--->
haveFun(gen ? makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(gen)
--->
haveFun(fun)
--->
tt
```

It shows how it simply requires start with `fun` to obtain `fun` at the end. Since this answer is not useful we look for the next one:

```
Maude> (next .)
Result: tt

Maude> (show path .)
haveFun(makeCalls(gen,gen))
--->
haveFun(gen ? makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(makeAnOffer(people,gen) ?
        makeCalls(makeAnOffer(people,makeAnOffer(people,gen))))
--->
haveFun(makeAnOffer(people,gen))
--->
haveFun(combine(gen,gen))
--->
haveFun(combine(burger,gen))
--->
haveFun(combine(burger,videogames))
--->
haveFun(fun)
--->
tt
```

In this case we would require to start having a `burger` and `videogames`, so they can be combined in order to reach the `fun`. In this case no friends were required. However, the next search (where we just show the last steps) requires a `burger`, `music`, and our friend `rober`:

```
...
haveFun(combine(gen,call(gen,gen)))
--->
haveFun(combine(burger,call(gen,gen)))
--->
haveFun(combine(burger,call(rober,gen)))
--->
haveFun(combine(burger,call(rober,music)))
--->
haveFun(combine(burger,videogames))
```

```
--->
haveFun(fun)
--->
tt
```

We can keep looking for more results until we find the one we are looking for or we reach the limit on the number of steps (which can be modified by means of the `depth` command).

4.2 Implementation notes

We have implemented our prototype in Maude [6], a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [16], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [4], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. This logic is a good semantic framework for formally specifying programming languages as rewrite theories [17]; since Maude specifications are executable, we obtain an interpreter for the language being specified.

Exploiting the fact that rewriting logic is reflective [7], an important feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [6, Chapter 14], a characteristic that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. In this way, we define the syntax of the modules introduced by the user, manipulate them, direct the evaluation of the terms (by using the on-demand strategy natural narrowing [10]), and implement the input/output interactions in Maude itself.

5 Concluding remarks and ongoing work

In this work we have proposed and formally proved the adequacy of a technique for lifting term rewriting derivations from an arbitrary instance of an expression to a constructed term—or the outer constructed part of any expression—using left-linear constructor systems. It is based on the generator technique from the field of functional-logic programming [8, 1], but adapted to the different semantic context of term rewriting [23]. For proving the adequacy of the proposed technique we have employed the semantics for constructor systems defined in [15] as the main technical tool. This way we have put the semantics in practice by using it for solving a technical problem that was not stated in the original paper. Along the way we have extended the semantics to support extra variables in rewriting rules, as those are very frequent when using narrowing, which is the context of the present paper. To do that we have made the necessary adjustments to the formulation of the semantics and to the proofs for its properties.

A fundamental limitation of the generators is that they can only be used for reaching ground c-terms or the outer constructed part of expressions. This limitation can be somewhat mitigated by reducing the reachability to a non-ground value to the reachability of a ground value: for example to test for $e \to^* c(X)$ we can define a new function f by the rule $f(c(X)) \to \mathit{true}$ and then test for $f(e) \to^* \mathit{true}$. Anyway this is a partial solution, and moreover the instantiation of free variables corresponding to the evaluation of *gen* cannot be obtained by a transformation in that line, for example by evaluating $(f(X),X)$ in the previous

example, due to the aforementioned loss of sharing between different occurrences of the same variable. This latter limitation could only be possibly overcomed by using some metaprogramming capabilities of the rewriting engine used to implement this technique. The generators technique has been used in practical systems, for example as the basis for an implementation of the functional-programming language Curry [5]. There the information provided by a Damas-Milner like type system is used to improve the efficiency, because instead of just one universal generator, like in our proposal, several generators are used, one for each type, which results in a great shrink of the search space for the evaluation of generators. One could argue that our generators are fundamentally equivalent to defining a generator *genE* that could be reduced to any expression, and then replacing each free or extra variable with *genE*, which would be trivially complete. Nevertheless, in our approach the search space for generators is significantly smaller, especially when combined with type information.

The system has been implemented in a Maude prototype that allows us to study their expressivity and possible applications. This prototype uses the on-demand strategy natural rewriting [10], thus providing an efficient implementation.

Regarding future work, we plan to improve our implementation by using the reflection capabilities of Maude to collect the evaluation of generators, in order to be able to present a computed answer for generators derivations, instead of relying on the trace to extract this information.

References

[1] Sergio Antoy & Michael Hanus (2006): *Overlapping Rules and Logic Variables in Functional Logic Programs*. In Sandro Etalle & Miroslaw Truszczynski, editors: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4079, Springer, pp. 87–101, doi:10.1007/11799573_9.

[2] Franz Baader & Tobias Nipkow (1998): *Term rewriting and all that*. Cambridge University Press, doi:10.1017/CBO9781139172752.

[3] Franz Baader & Wayne Snyder (2001): *Unification Theory*. In John Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning (in 2 volumes)*, Elsevier and MIT Press, pp. 445–532, doi:10.1016/B978-044450813-3/50010-2.

[4] Adel Bouhoula, Jean-Pierre Jouannaud & José Meseguer (2000): *Specification and proof in membership equational logic*. *Theoretical Computer Science* 236(1-2), pp. 35–132, doi:10.1016/S0304-3975(99)00206-6.

[5] Bernd Braßel, Michael Hanus, Björn Peemöller & Fabian Reck (2011): *KiCS2: A New Compiler from Curry to Haskell*. In Herbert Kuchen, editor: *Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings*, *Lecture Notes in Computer Science* 6816, Springer, pp. 1–18, doi:10.1007/978-3-642-22531-4_1.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer, doi:10.1007/978-3-540-71999-1.

[7] Manuel Clavel, José Meseguer & Miguel Palomino (2007): *Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic*. *Theor. Comput. Sci.* 373(1-2), pp. 70–91, doi:10.1016/j.tcs.2006.12.009.

[8] Javier de Dios Castro & Francisco Javier López-Fraguas (2007): *Extra Variables Can Be Eliminated from Functional Logic Programs*. *Electr. Notes Theor. Comput. Sci.* 188, pp. 3–19, doi:10.1016/j.entcs.2006.05.049.

[9] Francisco Durán, Steven Eker, Santiago Escobar, José Meseguer & Carolyn L. Talcott (2011): *Variants, Unification, Narrowing, and Symbolic Reachability in Maude 2.6*. In Manfred Schmidt-Schauß, editor: *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, *LIPIcs* 10, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 31–40, doi:10.4230/LIPIcs.RTA.2011.31.

[10] Santiago Escobar (2004): *Implementing Natural Rewriting and Narrowing Efficiently*. In Yukiyoshi Kameyama & Peter J. Stuckey, editors: *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings*, *Lecture Notes in Computer Science* 2998, Springer, pp. 147–162, doi:10.1007/978-3-540-24754-8_12.

[11] Michael Hanus (2005): *Functional Logic Programming: From Theory to Curry*. Technical Report, Christian-Albrechts-Universität Kiel.

[12] Jean-Marie Hullot (1980): *Canonical Forms and Unification*. In Wolfgang Bibel & Robert A. Kowalski, editors: *5th Conference on Automated Deduction, Les Arcs, France, July 8-11, 1980, Proceedings*, *Lecture Notes in Computer Science* 87, Springer, pp. 318–334, doi:10.1007/3-540-10009-1_25.

[13] H. Hussmann (1993): *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag.

[14] F. López-Fraguas, J. Rodríguez-Hortalá & J. Sánchez-Hernández (2009): *A Fully Abstract Semantics for Constructor Systems (Extended version)*. Technical Report SIC-2-09, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid.

[15] Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá & Jaime Sánchez-Hernández (2009): *A Fully Abstract Semantics for Constructor Systems*. In Ralf Treinen, editor: *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, *Lecture Notes in Computer Science* 5595, Springer, pp. 320–334, doi:10.1007/978-3-642-02348-4_23.

[16] José Meseguer (1992): *Conditioned Rewriting Logic as a Unified Model of Concurrency*. *Theor. Comput. Sci.* 96(1), pp. 73–155, doi:10.1016/0304-3975(92)90182-F.

[17] José Meseguer & Grigore Rosu (2007): *The rewriting logic semantics project*. *Theor. Comput. Sci.* 373(3), pp. 213–237, doi:10.1016/j.tcs.2006.12.018.

[18] José Meseguer & Prasanna Thati (2007): *Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols*. *Higher-Order and Symbolic Computation* 20(1-2), pp. 123–160, doi:10.1007/s10990-007-9000-6.

[19] Aart Middeldorp & Erik Hamoen (1994): *Completeness Results for Basic Narrowing*. *Appl. Algebra Eng. Commun. Comput.* 5, pp. 213–253, doi:10.1007/BF01190830.

[20] Juan Carlos González Moreno, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas & Mario Rodríguez-Artalejo (1999): *An Approach to Declarative Programming Based on a Rewriting Logic*. *J. Log. Program.* 40(1), pp. 47–87, doi:10.1016/S0743-1066(98)10029-8.

[21] A. Riesco & J. Rodríguez-Hortalá (2012): *Generators: Detailed proofs*. Technical Report 07/12, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid. Available at `http://gpd.sip.ucm.es/snarrowing`.

[22] Adrián Riesco & Juan Rodríguez-Hortalá (2012): *S-Narrowing for Constructor Systems*. In Abhik Roychoudhury & Meenakshi D'Souza, editors: *Theoretical Aspects of Computing - ICTAC 2012 - 9th International Colloquium, Bangalore, India, September 24-27, 2012. Proceedings*, *Lecture Notes in Computer Science* 7521, Springer, pp. 136–150, doi:10.1007/978-3-642-32943-2_10.

[23] Juan Rodríguez-Hortalá (2008): *A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems*. In Ramesh Hariharan, Madhavan Mukund & V. Vinay, editors: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India*, *LIPIcs* 2, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 328–339, doi:10.4230/LIPIcs.FSTTCS.2008.1764.

[24] L. Sterling & E. Shapiro (1986): *The Art of Prolog*. MIT Press, doi:10.1109/MEX.1987.4307074.