

S-Narrowing for Constructor Systems^{*}

Adrián Riesco and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
{ariesco, juanrh}@fdi.ucm.es

Abstract. Narrowing is a procedure that was conceived in the context of equational E-unification, and that has also 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 that has been obtained by using a normalized substitution can be ‘lifted’ to a narrowing derivation. Since then, 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.

In this work we propose a new narrowing notion for constructor systems that is based on the novel notion of s-unifier, that essentially allows a variable to be bound to several expressions at the same time. A Maude-based implementation for this narrowing relation, using an adaptation of natural narrowing as on-demand evaluation strategy, is presented, and its use for symbolic reachability analysis applied to the verification of cryptographic protocols is also outlined.

Keywords: narrowing, unification, constructor systems, Maude.

1 Introduction

Narrowing [3] is a procedure that was originally conceived in the context of equational E-unification, and that has also been used in a wide range of applications like for example symbolic reachability analysis [15], test-case generation [20], or as the basic operational mechanism of functional-logic languages [2]. Narrowing can be described as a modification of term rewriting in which matching is replaced by unification. By doing so, in a narrowing derivation from a starting goal expression, the narrowing procedure is able to deduce the instantiation of the variables of the goal expression that is needed for the computation to progress. This idea is reflected in *Hullot’s lifting lemma* [11], the key result for the completeness of narrowing w.r.t. term rewriting, which states that given an expression e_1 if we instantiate it with a substitution θ and we perform a term rewriting derivation $e_1\theta \rightarrow^* e_2$, then we can *lift* it into a narrowing derivation $e_1 \rightsquigarrow_{\sigma}^* e_3$ such that e_3 and σ are more general than e_2 and θ —w.r.t. to the usual instantiation preorder [4], and for the variables involved in the derivations—, provided that the starting substitution θ is normalized. This latter condition is essential: a

^{*} Research partially supported by the Spanish projects *DESAFIOS10* (TIN2009-14599-C03-01), *FAST-STAMP* (TIN2008-06622-C03-01/TIN), *PROMETIDOS-CM* (S2009/TIC-1465), and *GPD-UCM* (UCM-BSCH-GR58/08-910502).

normalized substitution only contains expressions in normal form in its range, which are expressions which cannot be reduced by term rewriting. It is fairly easy to break Hullot's lifting lemma by dropping that condition, for example under the term rewriting system (TRS) $\{f(0, 1) \rightarrow 2, \text{coin} \rightarrow 0, \text{coin} \rightarrow 1\}$, using the expression $f(X, X)$ and the non-normalized substitution $[X/\text{coin}]$ we can perform the term rewriting derivation $f(X, X)[X/\text{coin}] = f(\text{coin}, \text{coin}) \rightarrow f(0, \text{coin}) \rightarrow f(0, 1) \rightarrow 2$, which 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 [16,21,15,8].

In this paper we propose a new narrowing relation that tries to improve the completeness results for classic general narrowing, for the class of left-linear constructor-based term rewriting systems or just constructor systems (CS's). In particular we focus on dropping the normalization condition over the starting substitution that is required by Hullot's lifting lemma. In order to test the feasibility of the approach, we have implemented it in Maude [6]. The resulting prototype can be used to evaluate expressions with free variables under any given constructor system with extra variables.

Our starting point is a previous work [12], where a sound and complete compositional semantics for CS's was presented. CS's are characterized by having the signature partitioned in two disjoint sets of function symbols and constructor symbols, so any left-hand side of a rule has a function symbol in its root with constructed terms or just c-terms (expressions built using only constructor symbols and variables) as arguments, and no variable appears more than once in a left-hand side. CS's are usually used to represent programs in declarative languages, therefore we will use 'program' as a synonym for CS from now on. The semantics from [12] gives a characterization of the set of c-terms (an outer constructed part of any expression) reachable by term rewriting from expressions.¹ The key for getting compositionality in that semantics was using a suitable notion of semantic value. Instead of using c-terms, which may seem the obvious choice at a first look, a structured representation of the alternatives between c-terms in a term rewriting derivation is used so the constructor symbols have sets of values as arguments. For example, using a constructor symbol c with arity one and under the program $\{X ? Y \rightarrow X, X ? Y \rightarrow Y\}$ then $c(\{0, 1\})$ is a value for the expression $c(0 ? 1)$ but not for the expression $c(0) ? c(1)$, which reflects the different behavior of these expressions: if we add the rule $g(c(X)) \rightarrow d(X, X)$ to the program then it is easy to check that $g(c(0 ? 1)) \rightarrow^* d(0, 1)$ while $g(c(0) ? c(1)) \not\rightarrow^* d(0, 1)$, even though the set of c-terms reachable by $c(0 ? 1)$ and $c(0) ? c(1)$ is the same. These structured values are called s-terms, so an s-substitution or just s-csubst is any substitution with s-terms in its range. And as that semantics is compositional—in fact it is also fully abstract w.r.t. reachability of c-terms [12]—then any pair of expressions with the same set of s-terms are interchangeable in any context, as long as we are only concerned about the set of reachable c-terms. This is also reflected at the level of substitutions in an intermediate result of [12], that roughly states that if we can compute a value—i.e., reach that value/c-term by a term rewriting derivation—for an expression instantiated with an arbitrary substitution (for which normalization is not required), then we can

¹ We use the terminology *expression* instead of the more usual *term*—in the term rewriting community—in order to stress their difference with the more restricted notion of c-term.

compute the same value instantiating the same expression with an s -csubst such that every s -c-term in its range is a value for the corresponding expression in the range of the starting substitution. This makes sense because although an arbitrary substitution may implicitly contain an infinite amount of information in its range—as it may contain calls to functions with unbounded recursion—any finite term rewriting derivation is a finite computation process that therefore can only consume a finite amount of information in the form of values from the expressions in the range of that substitution. Note that in a sense we should consider that s -csubst are not normalized because they contain alternatives between expressions, so we could evaluate any s -csubst to several c -subst by choosing an element in each of the sets that appear in the s -c-terms in the range of the s -csubst.

But what it is important for our purpose here is that this result shows that, for reachability of c -tems in CS's, s -csubstitutions have the same power as arbitrary substitutions. And that is good because narrowing derivations use the left-hand sides of program rules to deduce the instantiation of variables in the goal expression needed for the computation to progress, by syntactic unification in the case of classic narrowing. But we have seen that, in order to have the same power as arbitrary substitutions, what we need to deduce from those left-hand sides is an s -csubst, instead of a normalized substitution. To do that we propose a modification of a classical syntactic unification algorithm that now allows a variable to be bound to several expressions at the same time. We use this novel *s-unification* algorithm as the basis to define a new narrowing relation called *s-narrowing*, that gathers up all the c -terms to which a variable has been bound during the computation. Doing so for every variable in the starting goal expression, and also for the variables in the expressions it has been bound to, we end up building the s -csubst that solves the goal. Applying these ideas to lift the derivation from the example above we get the following s -narrowing derivation:

$$f(X, X) \mid \emptyset \rightsquigarrow 2 \mid \{X \mapsto \{0, 1\}\}$$

where the following successful s -unification derivation is used in the application of the rule for f .

$$\begin{aligned} & \{f(0, 1) \stackrel{?}{=} f(X, X)\}; \emptyset \Rightarrow \{0 \stackrel{?}{=} X, 1 \stackrel{?}{=} X\}; \emptyset \Rightarrow \{X \stackrel{?}{=} 0, 1 \stackrel{?}{=} X\}; \emptyset \\ & \Rightarrow \{1 \stackrel{?}{=} X\}; \{X \mapsto \{0\}\} \Rightarrow \{X \stackrel{?}{=} 1\}; \{X \mapsto \{0\}\} \Rightarrow \emptyset; \{X \mapsto \{0, 1\}\} \end{aligned}$$

Regarding the prototype, s -narrowing is implemented by using an adaptation of the natural narrowing on-demand strategy [9], which indicates the positions that must be reduced in each step. As a proof-of-concept we have tested the prototype with several examples, including a sketch of the verification of cryptographic protocols.

The rest of the paper is organized as follows. In Section 2 we explain the aforementioned semantics for constructor systems, and use it to formalize the intuitions presented in the introduction. In Section 3 we present the notions of s -unification and s -narrowing and some interesting results about them. Then in Section 4 we outline the implementation and commands of our prototype using examples. Finally, Section 5 concludes and outlines some lines of future work. More information and detailed proofs of the results shown here are presented in [18].

2 Preliminaries and Formal Setting

2.1 Basic Syntax

We consider a first order signature $\Sigma = CS \uplus FS$, where CS and FS are two disjoint set of *constructor* and defined *function* symbols respectively, all them with associated arity. We write CS^n (FS^n resp.) for the set of constructor (function) symbols of arity n . We write c, d, \dots for constructors, f, g, \dots for functions and X, Y, \dots for variables of a numerable set \mathcal{V} . The notation \bar{o} stands for tuples of any kind of syntactic objects.

The set *Exp of expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \dots, e_n)$, where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \dots, e_n \in Exp$. The set *CTerm of constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). We will write e, e', \dots for expressions and t, s, \dots for c-terms. The set of variables occurring in an expression e will be denoted as $var(e)$. We say that an expression e is *ground* iff $var(e) = \emptyset$. We will frequently use *one-hole contexts*, defined as $Cntxt \ni C ::= [] \mid h(e_1, \dots, C, \dots, e_n)$, with $h \in CS^n \cup FS^n$. The application of a context C to an expression e , written by $C[e]$, is defined inductively as $[] [e] = e$ and $h(e_1, \dots, C, \dots, e_n)[e] = h(e_1, \dots, C[e], \dots, e_n)$.

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

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathcal{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ε for the identity (or empty) substitution. We write $e\theta$ for the application of θ to e , and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and range of θ are defined as $dom(\theta) = \{X \in \mathcal{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. By $[X_1/e_1, \dots, X_n/e_n]$ we denote a substitution σ such that $dom(\sigma) = \{X_1, \dots, X_n\}$ and $\forall i. \sigma(X_i) = e_i$. Similarly the notation $[X/e \mid P(X, e)]$ where P is some predicate over X and e is used to define substitutions using a set-like notation, so $([X/e \mid P(X, e)])(Y) = e'$ if $P(Y, e')$, and $([X/e \mid P(X, e)])(Y) = Y$ otherwise. 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 $W \subseteq \mathcal{V}$ 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 $\theta|_{\mathcal{V} \setminus D}$ as a shortcut for $\theta|_{(\mathcal{V} \setminus D)}$. *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.

A constructor-based term rewriting system or just *constructor system* or *program* \mathcal{P} (CS) is a set of c-rewrite rules of the form $f(\bar{t}) \rightarrow r$ 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 $X \in \mathcal{V}$ is an extra variable in the rule $l \rightarrow r$ iff $X \in var(r) \setminus var(l)$. the set of extra variables in a program rule R . A fresh variant of a program rule is the result of taking a program rule and applying to it a substitution that replaces each variable of the rule by a fresh variable. We assume that every CS contains the rules $Q = \{X ? Y \rightarrow X, X ? Y \rightarrow Y\}$, defining the behavior of $?_ \in FS^2$, used 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 \mathcal{P} , its associated *term rewriting relation* $\rightarrow_{\mathcal{P}}$ is defined as: $C[l\sigma] \rightarrow_{\mathcal{P}} C[r\sigma]$ for any context C , rule $l \rightarrow r \in \mathcal{P}$ and $\sigma \in Subst$. We write $\rightarrow_{\mathcal{P}}^*$ for the reflexive and transitive closure of the relation $\rightarrow_{\mathcal{P}}$. In the following, we will usually omit the reference to \mathcal{P} or denote it by $\mathcal{P} \vdash e \rightarrow e'$ and $\mathcal{P} \vdash e \rightarrow^* e'$. By $\mathcal{P} \vdash e_1 \downarrow e_2$ we denote that e_1 and e_2 are *joinable* under \mathcal{P} , i.e., it exists some expression e_3 such that $\mathcal{P} \vdash e_1 \rightarrow^* e_3 \leftarrow^* e_2$, where \leftarrow denotes the inverse of \rightarrow , and \leftarrow^* the reflexive-transitive closure of \leftarrow .

2.2 A Proof Calculus for Constructor Systems with Extra Variables

In [12] an adequate semantics for reachability of c-terms by term rewriting in CS's was presented. As we mentioned in Section 1, the key idea in that semantics is using a suitable notion of value, in this case the notion of s-c-term, which is a structured representation of alternative between c-terms in a term rewriting derivation. An s-c-term is a *finite* set of elemental s-c-terms, that are variables or constructors applied to s-c-terms, so *SCTerm* is an alias for the set of finite sets of elemental s-c-terms and the set *ESCTerm* of elemental s-c-terms is defined as $ESCTerm \ni est ::= X \mid c(st_1, \dots, st_n)$ for $X \in \mathcal{V}$, $c \in DC^n$, $st_1, \dots, st_n \in SCTerm$. 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 for s-expressions \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 $X \in \mathcal{V}$, and $h(se_1, \dots, se_n) \sqsubseteq h(se'_1, \dots, se'_n)$ iff $\forall i. se_i \sqsubseteq se'_i$.

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-c-terms, respectively. Some care must be taken when extending s-substs to be applied to *ESExp* and *SExp*, so for any $\sigma \in SSubst$ we define $\sigma : ESExp \rightarrow SExp$ as $X\sigma = \sigma(X)$, $h(\overline{se})\sigma = \{h(\overline{\sigma se})\}$; and $\sigma : SExp \rightarrow SExp$ as $se\sigma = \bigcup_{ese \in se} ese\sigma$. The approximation preorder \sqsubseteq is defined for s-substs as $\sigma \sqsubseteq \theta$ iff $\forall X \in \mathcal{V}. \sigma(X) \sqsubseteq \theta(X)$.

In this semantics the denotation of an expression is obtained as the denotation of its associated s-expression, assigned by the operator $\tilde{_} : Exp_{\perp} \rightarrow SExp$, which is defined as

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

Fig. 1. A proof calculus for constructor systems

$\tilde{\perp} = \emptyset$; $\tilde{X} = \{X\}$ for any $X \in \mathcal{V}$; $\widetilde{h(e_1, \dots, e_n)} = \{h(\tilde{e}_1, \dots, \tilde{e}_n)\}$ for any $h \in \Sigma^n$. The operator $\tilde{\cdot}$ is extended to s-substitutions as $\tilde{\sigma}(X) = \sigma(X)$, for $\sigma \in Subst_{\perp}$. It is easy to check that $\tilde{e\sigma} = \tilde{e}\tilde{\sigma}$ (see [12]). Conversely, we can flatten an s-expression se to obtain the set $flat(e)$ 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 $flat(X) = \{X\}$; $flat(h(se_1, \dots, se_n)) = \{h(e_1, \dots, e_n) \mid e_i \in flat(se_i) \text{ for } i = 1..n\}$.

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 $se \rightarrow st$ 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 [12] for a detailed explanation of the intuitions behind the rules of the calculus. We write $\mathcal{P} \vdash se \rightarrow st$ to express that $se \rightarrow st$ is derivable in our calculus under the CS \mathcal{P} . The *denotation* of an s-expression se under a CS \mathcal{P} is defined as $\llbracket se \rrbracket^{\mathcal{P}} = \{st \in SCTerm \mid \mathcal{P} \vdash se \rightarrow st\}$. In the following we will usually omit the reference to \mathcal{P} . The denotation of an s-substitution σ is defined as $\llbracket \sigma \rrbracket = \{\theta \in SCSubst \mid \forall X \in \mathcal{V}, \sigma(X) \rightarrow \theta(X)\}$.

The setting originally presented in [12] was not able to deal with extra variables, but in [17] we extended it to deal with them, which is just needed for the present work, as extra variables are very common when using narrowing. To do that we were not required to change the rules of the calculus, but only the proof for the adequacy, as the rule **ROR** from Figure 1 already allows to instantiate extra variables freely with s-terms. Nevertheless, as a consequence of that freely instantiation of extra variables, every program with extra variables turns into non-deterministic. For example consider a program $\{f \rightarrow (X, X)\}$ for which the constructors $0, 1 \in CS^0$ are available, then we can prove $\tilde{f} = \{f\} \rightarrow \{\{\emptyset\}, \{1\}\} = \widetilde{(0, 1)}$. 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 [19] induced by term rewriting. As a consequence of this—as seen in Section 2.1—we assume that all the programs contains the function $?$ defined by the rules $Q = \{X ? Y \rightarrow X, X ? Y \rightarrow Y\}$, 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. Anyway, the point

is that having $\hat{?}$ at one's disposal is enough to express the non-determinism of any program [10], so we can use it to define the transformation $\hat{\sqsubset}$ from s-exp and elemental s-exp to partial expressions that, contrary to *flat*, now takes care of keeping the nested set structure by means of uses of the $\hat{?}$ function. Then $\hat{\sqsubset}: ESExp \rightarrow Exp_{\perp}$ is defined by $\hat{X} = X$, $\widehat{h(se_1, \dots, se_n)} = h(\widehat{se_1}, \dots, \widehat{se_n})$; and $\hat{\sqsubset}: SExp \rightarrow Exp_{\perp}$ is defined by $\widehat{\emptyset} = \perp$, $\widehat{\{ese_1, \dots, ese_n\}} = \widehat{ese_1} \hat{?} \dots \hat{?} \widehat{ese_n}$ for $n > 0$, where in the case for $\{ese_1, \dots, ese_n\}$ we use some fixed arbitrary order on terms for arranging the arguments of $\hat{?}$. This operator is also overloaded for substitutions as $\hat{\sqsubset}: SSubst \rightarrow Subst_{\perp}$ as $\widehat{(\sigma)}(X) = \sigma(\widehat{X})$. Thanks to the power of $\hat{?}$ to express non-determinism, that transformation preserves the semantics from Figure 1, so the following result can be proved—see [12] for details about the proof.

Theorem 1 (Adequacy of $\llbracket - \rrbracket$). *For all $e, e' \in Exp, t \in CTerm_{\perp}, st \in SCTerm$:*

Soundness *$st \in \llbracket \tilde{e} \rrbracket$ and $t \in flat(st)$ implies $e \rightarrow^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Therefore, $\tilde{t} \in \llbracket \tilde{e} \rrbracket$ implies $e \rightarrow^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Besides, in any of the previous cases, if t is total then $e \rightarrow^* t$.*

Completeness *$e \rightarrow^* e'$ implies $\tilde{|e'|} \in \llbracket \tilde{e} \rrbracket$. Hence, if t is total then $e \rightarrow^* t$ implies $\tilde{t} \in \llbracket \tilde{e} \rrbracket$.*

We conclude this section with the following result, that formalizes the intuitions we gave in Section 1 stating that we only need to compute an s-csubst in order to lift any term rewriting derivation starting from an expression instantiated with an arbitrary substitution, if we only care about reachability of c-terms—or its outer constructed part, expressed by the notion of shell.

Proposition 1. *For all $e, e' \in Exp, \sigma \in Subst, e\sigma \rightarrow^* e'$ implies $\exists \theta \in \llbracket \sigma \rrbracket. e\hat{\theta} \rightarrow^* e''$ such that $|e'| \sqsubseteq |e''|$. Note that $\theta \in \llbracket \sigma \rrbracket$ implies $\theta \in SCSubst$. Besides, if $e' = t \in CTerm$ then $e\hat{\theta} \rightarrow^* t$.*

3 S-Narrowing and S-Unification

In this section we will present our proposal for the novel s-narrowing relation—where ‘s’ stands for “set,” as in s-cterm—in which we realize the ideas about a new narrowing relation discussed in Section 1. As suggested by Proposition 1, in s-narrowing we use the information contained in the left-hand sides of program rules to compute an s-csubst, in order to lift any term rewriting derivation starting from the instantiation of an expression with an arbitrary substitution. To do that we rely on the notion of s-unification, a modification of syntactic unification that basically allows a variable to be bound to several expressions at the same time.

For the sake of conciseness of the notation, in the rest of the paper we will often omit the braces in singleton sets, so the context determines whether e refers to $\{e\}$ —as $\{0\}$ in $c(0) \in SExp$ —or just to e —as 0 in $c(0) \in Exp$.

3.1 S-Unification

The main difference between s-unification and syntactic unification is that, instead of finding a substitution that makes two expressions equal, in s-unification we look for an

VTRIV	$\{X \stackrel{?}{=} X\} \uplus P; S \Rightarrow P; S$	if $X \in \mathcal{V}$
DEC	$\{h(e_1, \dots, e_n) \stackrel{?}{=} h(e'_1, \dots, e'_n)\} \uplus P; S \Rightarrow \{e_1 \stackrel{?}{=} e'_1, \dots, e_n \stackrel{?}{=} e'_n\} \uplus P; S$	
CLASH	$\{h_1(\overline{e_1}) \stackrel{?}{=} h_2(\overline{e_2})\} \uplus P; S \Rightarrow fail$	if $h_1 \neq h_2$
TURN	$\{e \stackrel{?}{=} X\} \uplus P; S \Rightarrow \{X \stackrel{?}{=} e\} \uplus P; S$	if $e \notin \mathcal{V}$
ADDBIND	$\{X \stackrel{?}{=} e\} \uplus P; S \Rightarrow P; S \oplus \{X \mapsto e\}$	

Fig. 2. S-Unification algorithm \mathcal{S}

s-subst that makes the intersection of two expressions a nonempty set. From the term rewriting point of view this means that an s-unifier of two expressions makes them joinable. Formally, $\sigma \in SSubst$ is an s-unifier of $e_1, e_2 \in Exp$ iff $Q \vdash e_1 \hat{\sigma} \downarrow e_2 \hat{\sigma}$. A particularity of s-unification is that occurs check is not needed: for example we can instantiate the expressions X and $c(X)$ so they have a nonempty intersection by using $[X/\{X, c(X)\}]$, as $Q \vdash X[\overline{X/\{X, c(X)\}}] = X ? c(X) \rightarrow c(X) \leftarrow c(X) ? c(X) = c(X)[\overline{X/\{X, c(X)\}}]$.

In Figure 2 we formulate our rule-based s-unification algorithm \mathcal{S} , following the style of the rule-based algorithm \mathcal{U} from [4] for computing the most general syntactic unifier. Hence, in \mathcal{S} we rewrite configurations of the shape $P; S$ where P is the problem, i.e., a finite set of equations of the shape $e_1 \stackrel{?}{=} e_2$ between the expressions to unify, and S is the solution computed so far, represented as a finite set of bindings of the shape $X \mapsto \{e_1, \dots, e_n\}$ for $X \in \mathcal{V}$ and $e_1, \dots, e_n \in Exp$. The special configuration *fail* is used to indicate a failure in the s-unification process. Given a solution S , its domain $dom(S)$ is the set of variables for which a binding is defined in S . By $S[X]$ we denote the binding corresponding to X in S , and by $S[X \mapsto s]$ we denote the solution S' such that $S'[X] = s$ and $S'[Y] = S[Y]$ for each $Y \in dom(S) \setminus \{X\}$. The operator \oplus is used to add a new element to the binding for a variable in a solution, and it is defined as $S \oplus \{X \mapsto e\} = S[X \mapsto \{e\}]$ if $X \notin dom(S)$; $S[X \mapsto c] \oplus \{X \mapsto e\} = S[X \mapsto c \cup \{e\}]$ otherwise. Given some $\mathcal{W} \subseteq \mathcal{V}$ by $S|_{\mathcal{W}}$ we denote the restriction of S to \mathcal{W} , i.e., the result of dropping from S the bindings for variables which are not contained in \mathcal{W} ; and by $S|_{\mathcal{V} \setminus \mathcal{W}}$ we denote $S|_{(\mathcal{V} \setminus \mathcal{W})}$.

In order to s-unify two given expressions $e_1, e_2 \in Exp$ we start with $\{e_1 \stackrel{?}{=} e_2\}; \emptyset$ as the initial configuration and apply the rules of \mathcal{S} in a don't care non-deterministic fashion until reaching *fail* or a configuration of the shape $\emptyset; S$, which is a configuration in solved form. By \Rightarrow^* we denote the reflexive-transitive closure of \Rightarrow , therefore $\{e_1 \stackrel{?}{=} e_2\}; \emptyset \Rightarrow^* \emptyset; S$ indicates that the s-unification procedure for e_1 and e_2 has ended with success computing the solution S . The rules VTRIV, DEC, CLASH and TURN are standard in unification algorithms. The novelty in \mathcal{S} compared to \mathcal{U} is the rule ADDBIND that, together with the absence of a rule for occurs check, tries to reflect the intended meaning of an s-unifier discussed above. Maybe the reader could expect a special case for occurs check where a binding $X \mapsto \{X, e\}$ would be added to the solution, but that case is not needed because of the way we interpret the solutions computed by \mathcal{S} , as we will see below.

We conjecture that the set of pairs of expressions that are s-unifiable is bigger than the set of pairs of expressions that are unifiable. However, the algorithm \mathcal{S} only

grants the absence of cycles in the computed solutions when unifying pairs of expression e_1 and e_2 such that $\text{var}(e_1) \cap \text{var}(e_2) = \emptyset$ and e_1 is linear, which is enough for its uses in s-narrowing. Otherwise the computed solution may contain cyclic bindings: consider for example the problem $h(X, Z, Y) \stackrel{?}{=} h(Z, Y, X)$ which is unifiable with $[Z/Y, X/Y]$, and for which S computes the cyclic solution $[X/Z, Z/Y, Y/X]$; or the problem $d(X, c(X)) \stackrel{?}{=} d(Y, Y)$ which is not unifiable but for which S computes the cyclic solution $[X/Y, Y/c(X)]$, even though $d(X, c(X))$ and $d(Y, Y)$ do not share variables. The absence in S of a rule for variable elimination, that would propagate the binding computed for one variable to the rest of the problem, allows us for example to s-unify $d(X, X)$ and $d(0, 1)$ with $[X/\{0, 1\}]$. But, at the same time, it implies that sometimes S will not compute the most general unifier for two unifiable expressions, so it is not a conservative extension of a unification algorithm. For example $f(c(U), c(V)) \stackrel{?}{=} f(X, X); \emptyset \Rightarrow^* \emptyset; \{X \mapsto \{c(U), c(V)\}\}$, while $[X/c(U), V/U]$ is the most general unifier of $f(c(U), c(V))$ and $f(X, X)$. In order to be more conservative, we could have opted for an alternative definition of the rule ADDBIND in which the bindings computed so far would be reused. But, as we will see in Section 3.2, that would entail computing an s-narrowing solution that would be more concrete than what is needed to lift the term rewriting derivations, so we use ADDBIND as defined above. The algorithm S is terminating as shown in the following result, in the line of [4].

Proposition 2. *For any problem P , every sequence $P; \emptyset \Rightarrow P_1; S_1 \Rightarrow P_2; S_2 \Rightarrow \dots$ terminates either with fail or with a configuration of the shape $\emptyset; S$*

By $S^*[X]$ we denote the binding corresponding to X in S after resolving the indirections caused by variables in $S[X]$ that are also in the domain of S , which is defined as $S^*[X] = (S[X])[\overline{Y}/S^*[\overline{Y}]]$ for $\overline{Y} = \text{var}(S[X]) \cap \text{dom}(S)$. Hence in general $S^*[X] \in \text{SExp}$. Note that $S^*[X]$ is only well defined for solutions S without cyclic bindings, but that is enough for us as we will only deal with solutions with acyclic bindings. Using this notion we define the *SSubst corresponding to a solution S* , denoted by σ_S , as $\sigma_S = [X/S^*[X] \mid X \in \text{dom}(S)]$. Although we do not provide a formal proof, we conjecture that if $\text{var}(e_1) \cap \text{var}(e_2) = \emptyset$ and e_1 is linear then $\{e_1 \stackrel{?}{=} e_2\}; \emptyset \Rightarrow^* \emptyset; S$ implies that σ_S° is an s-unifier of e_1 and e_2 , where the opening σ° of an s-subst σ is defined as $(\sigma^\circ)(X) = \{X\} \cup \sigma(X)$. In fact, in s-unification and s-narrowing we treat any substitution and its opening as if they were indistinguishable, which reflects a view of variables as ever fruitful sources of c-terms. In s-narrowing free variables are never really instantiated, but different alternative binding for the variables are collected, hence a variable can always “be itself” again when needed, so it can be bound to a c-term it was not previously bound.

3.2 S-Narrowing

The s-narrowing relation is defined in Figure 3. In s-narrowing we work with configurations of the shape $e \mid S$ where e is a goal expression and S is a solution like those used in s-unification. We do this in order to avoid instantiating the variables in the goal, so we could bind them to several c-terms at the same time. In this way, we collect in S the bindings for those variables. The idea of s-narrowing is pretty simple. First we s-unify

$$\begin{array}{l}
C[f(\bar{e})] \mid S_1 \rightsquigarrow C[r\sigma_p] \mid S_2 \text{ for any fresh variant } (f(\bar{p}) \rightarrow r) \in \mathcal{P} \text{ such that:} \\
i) \{f(\bar{p}) \stackrel{?}{=} f(\bar{e})\}; S_1 \Rightarrow^* \emptyset; S \quad ii) S_p = S|_{\text{var}(\bar{p})} \text{ and } S_2 = S|_{\setminus \text{var}(\bar{p})} \quad iii) \sigma_p = \widehat{\sigma_{S_p}}
\end{array}$$

Fig. 3. S-Narrowing

an expression $f(\bar{e})$ occurring in the goal expression with the left hand-side of a fresh variant of a program rule. To do that we start s-unification using the solution computed so far, that contains the bindings collected for the goal subexpression. As the variant is fresh then occurs check is not needed. If s-unification succeeds then we take the part S_p of the solution corresponding to the fresh left-hand side and use it for parameter passing. The following result ensures that each s-expressions in the range of S_p is singleton, so $\widehat{\sigma_{S_p}} = \sigma_p \in \text{Subst}$:

Lemma 1. *For all $e_1, e_2 \in \text{Exp}$ if e_1 is linear, $\text{var}(e_1) \cap \text{var}(e_2) = \emptyset$ and $\{e_1 \stackrel{?}{=} e_2\}; \emptyset \Rightarrow^* \emptyset; S$, then for $S_{e_1} = S|_{\text{var}(e_1)} \forall X \in \text{dom}(S_{e_1}) S_{e_1}[X]$ is singleton.*

Then the propagation of the bindings computed for $f(\bar{e})$ is implicitly performed by using S_2 in the resulting s-narrowing configuration, as it is the part of the solution for the s-unification that does not affect the fresh left-hand side. By \rightsquigarrow^* we denote the reflexive-transitive closure of \rightsquigarrow . A successful s-narrowing derivation for an expression e is a derivation $e \mid \emptyset \rightsquigarrow^* t \mid S$ where t is a c-term. Then, similarly to s-unification, the s-subst computed as solution by that s-narrowing derivation is σ_S° .

Note that the application of σ_p to r is needed to ensure soundness, as we can see considering the program $\{f(X) \rightarrow g(X), g(1) \rightarrow 2\}$. If we drop the application of σ_p at each step, then we can do:

$$\begin{array}{l}
f(0) \mid \emptyset \rightsquigarrow g(X_1) \mid \{X_1 \mapsto \{0\}\} \quad \text{as } \{f(X_1) \stackrel{?}{=} f(0)\}; \emptyset \Rightarrow^* \emptyset; \{X_1 \mapsto \{0\}\} \\
\rightsquigarrow 2 \mid \{X_1 \mapsto \{0, 1\}\} \quad \text{as } \{g(1) \stackrel{?}{=} g(X_1)\}; \{X_1 \mapsto \{0\}\} \Rightarrow^* \emptyset; \{X_1 \mapsto \{0, 1\}\}
\end{array}$$

but this is clearly unsound because $f(0)[X_1/0?1] \not\rightsquigarrow^* 2$, and in fact there is no $\sigma \in \text{Subst}$ such that $f(0)\sigma \rightarrow^* 2$. Thus the application of σ_p is necessary to respect the restrictions imposed by the symbols of Σ present in the goal expression which, contrary to variables, cannot be replaced by the application of substitutions. Conversely, if we use \rightsquigarrow as defined in Figure 3 then the derivation gets stuck after the first step, as expected:

$$f(0) \mid \emptyset \rightsquigarrow g(0) \mid \emptyset \quad \text{as } \{f(X_1) \stackrel{?}{=} f(0)\} \mid \emptyset \Rightarrow^* \emptyset \mid \{X_1 \mapsto \{0\}\}$$

and we cannot continue as $\{g(1) \stackrel{?}{=} g(0)\}; \emptyset \Rightarrow \{1 \stackrel{?}{=} 0\}; \emptyset \Rightarrow \text{fail}$. Just like classical narrowing can be rephrased as a unification step followed by a term rewriting step, i.e. as $C[f(\bar{e})] \Rightarrow C\sigma[f(\bar{e})\sigma] \rightarrow C\sigma[r\sigma]$, we could similarly rephrase s-narrowing as $C[f(\bar{e})] \mid S_1 \Rightarrow C[f(\bar{e})\sigma_p] \mid S_2 \rightarrow C[r\sigma_p] \mid S_2$.

The following example shows why we open the substitution computed as solution. Given the program $\{f(0, 1, X) \rightarrow X, \text{coin} \rightarrow 0, \text{coin} \rightarrow 1\}$ and the goal $f(X, X, X)$ for which we can compute $f(X, X, X) \mid \emptyset \rightsquigarrow X \mid \{X \mapsto \{0, 1\}\}$. If we use σ_S with $S = \{X \mapsto \{0, 1\}\}$ as the computed solution then we could not reach X by term rewriting, as $f(X, X, X)[X/\{0, 1\}] \not\rightsquigarrow^* X$, while we can reach it using the non-normalized

substitution $[X/\text{coin} ? X]$. But if we open the solution and use σ_S° as the computed solution, as we originally proposed, then $f(X, X, X) \widehat{[X/\{0, 1\}]^\circ} \rightarrow^* f(0, 1, X) \rightarrow X$. This is also coherent with our view of free variables in s-narrowing, which are never instantiated and are always implicitly bound to (the singleton set containing) themselves.

In Section 3.1 we saw that s-unification is not a conservative extension of unification because the bindings in the solution computed so far are not reused to solve subsequent equations. The following example illustrates how reusing those bindings would result in computing too specific solutions. Consider the program $\{f(c(X), Y) \rightarrow h(X, Y), h(X, c(Y)) \rightarrow g(X, Y), g(0, 1) \rightarrow 2\}$ and the goal expression $f(X, X)$, for which we can do $f(X, X) \mid \emptyset \rightsquigarrow h(U, X) \mid \{X \mapsto c(U)\}$ —for the sake of conciseness we drop the bindings for irrelevant variables. Now in order to unify $h(U, X)$ with $h(W, c(V))$ —a fresh variant of the left-hand side of the rule for h —we have two options. On the one hand, if we modify the rule `ADDBIND` in order to reuse the binding in $\{X \mapsto c(U)\}$ then we can perform the step $h(U, X) \mid \{X \mapsto c(U)\} \rightsquigarrow g(U, U) \mid \{X \mapsto c(U)\}$ and then $g(U, U) \mid \{X \mapsto c(U)\} \rightsquigarrow 2 \mid \{X \mapsto c(U), U \mapsto \{0, 1\}\}$, thus getting the solution $[X/c(\{0, 1\})]^\circ$ —for conciseness here we restrict the solution to the variables in the starting goal. On the other hand, if we use the proposed definition of s-unification then derivation proceeds as $h(U, X) \mid \{X \mapsto c(U)\} \rightsquigarrow g(U, V) \mid \{X \mapsto \{c(U), c(V)\}\} \rightsquigarrow 2 \mid \{X \mapsto \{c(U), c(V)\}, U \mapsto 0, V \mapsto 1\}$, getting the solution $[X/\{c(0), c(1)\}]^\circ$. Although both solutions are sound in the sense that both $f(X, X) \widehat{[X/c(\{0, 1\})]^\circ} \rightarrow^* 2$ and $f(X, X) \widehat{[X/\{c(0), c(1)\}]^\circ} \rightarrow^* 2$, the solution computed by the original definition is better in the sense that $[X/\{c(0), c(1)\}]^\circ \sqsubseteq [X/c(\{0, 1\})]^\circ$ while $[X/c(\{0, 1\})]^\circ \not\sqsubseteq [X/\{c(0), c(1)\}]^\circ$. This is also reflected at the term rewriting level, as seen with function g in Section 1. For these reasons we have chosen not to reuse bindings in s-unifications.

We have not obtained any formal result about the adequacy of s-narrowing yet, so we only have some conjectures. Regarding soundness, we think that $e_1 \mid S_1 \rightsquigarrow^* e_2 \mid S_2$ implies $e_1 \widehat{\sigma_{S_2}^\circ} \rightarrow^* e_2$. For completeness we would like to prove a lifting lemma in the style of Hullot's one, but first we have to find an appropriate order to be used there. That is pretty difficult because that order should be able to express at the same time that the computed substitution neither instantiates too much, nor introduces redundant alternatives in the sets contained in the s-expressions in its range. Therefore it would be a combination of the usual instantiation preorder [4] and the preorder \sqsubseteq , that also should treat any expression and its opening as equivalent. Hence, a lot of additional work should be put in developing the theory of s-unification.

4 Maude Prototype and Sample Application

We present in this section our prototype and outline its implementation. Much more information can be found at <http://gpd.sip.ucm.es/snarrowing>.

4.1 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* [13], 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* [5], 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 [14]; since Maude specifications are executable, we obtain an interpreter for the language being specified.

Exploiting the fact that rewriting logic is reflective, 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 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 on-demand strategies), and implement the input/output interactions in Maude itself.

An important point of our implementation is the use of an adaptation of the on-demand evaluation strategy natural narrowing [9], which generates a matching definitional trees for each function symbol and then traverses them to decide the position of the current term where narrowing must be applied. However, the description of natural narrowing presented in [9] used syntactic unification while traversing the definitional trees used by the technique, which leads to incompleteness in our approach. For this reason we have slightly modified the algorithm to use s-unification, which implies modifying the application of the unifier to the current term in order to preserve matching.

4.2 Prototype

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 STMENTS ends`, where `NAME` is the identifier of the program and `STMENTS` is a sequence of constructor-based left-linear rewrite rules, written in the following format:

```
(smod ICTAC is
  f(c(X),Y) -> h(X,Y) .
  h(X, c(Y)) -> g(X,Y) .
  g(0,1) -> 2 . ends)
```

where upper-case letters are assumed to be variables. We can first see how the tool solves s-unification problems with the `=?` command:

```
Maude> (g(0, 1) =? g(X,X) .)
X -> 0 ? 1
```

We can evaluate terms with variables by using s-narrowing with the natural narrowing strategy, which is used with the command:

```
Maude> (narrowing f(X,X) .)
{2, X -> c(0) ? c(1)}
```

The narrowing command returns the obtained result (2 in this case) as well as the required substitution, that in this case indicates that the variable X must take the set of values composed of $c(0)$ and $c(1)$. We can ask the system for more solutions with the `cont` command until no more solutions (as in the current example) are found:

```
Maude> (cont .)
No more solutions.
```

Finally, the system combines the on-demand techniques, that indicate the positions and the rules that must be used, with two different search strategies: `depth-first` and `breadth-first`. These strategies can be switched with `breadth-first` and `depth-first`.

4.3 The Dolev-Yao Intruder Model Using S-Narrowing

We present an implementation of the Dolev-Yao intruder model [7] in the line of [15] but now using *s*-narrowing, as proof-of-concept of our system. Note that the different features provided but these languages make the implementation rather different. We first define `alice` and `bob` as the possible roles or participants:

```
roles -> alice .
roles -> bob .
```

Decryption of messages is specified by using a ground simulation of the equality constraint, where we use the constructor `enc` to define the encryption of messages and `inv` as data constructor for inverting a key:

```
decrypt(enc(M,k1),inv(k1)) -> M .
decrypt(enc(M,k2),inv(k2)) -> M .
```

The `protocol` function associates to each participant a set of actions, which are the answers he or she returns for a given question. First, `alice` share a pair with the messages `ma1` and `ma2`, using the key received as parameter to encrypt them. Note that *the same parameter is used in both messages*:

```
protocol(alice, X) -> p(enc(ma1, X), enc(ma2, X)) .
```

When `bob` receives the message `ma1` encrypted with the `k1` key he sends `mb1`; similarly, he sends `mb2` when he receives `ma2` encrypted with `k2`. In these rules and the one above lies the novelty of the *s*-narrowing approach: the variable X above must be bound to both `k1` and `k2` for `bob` to send the appropriate messages:

```
protocol(bob, enc(ma1, k1)) -> mb1 .
protocol(bob, enc(ma2, k2)) -> mb2 .
```

Finally, if `alice` receives a pair with the two messages from `bob` she sends the inverse of `k1`, that can be used to decrypt, for example, `enc(ma1, k1)`:

```
protocol(alice, p(mb1, mb2)) -> inv(k1) .
```

The function `discover` models the messages that can be deduced by the intruder from a starting set of messages, where `discStep` combines the information generated by the responses of `alice` and `bob` to the queries of the intruder, and the one generated by the intruder by combining the starting messages according to the Dolev-Yao model:

```
discover(M) -> M ? discover(discStep(M) ? M) .
discStep(M) -> protocol(roles, M) ? dyStep(M) .
```

The auxiliary `dyStep` function can generate pairs of the elements, split these pairs, encrypt, and decrypt, thus representing the recombination of information the intruder is able to perform, according to the Dolev-Yao model for the intruder capabilities. Note that in this function the same variable `M` appears twice in the right-hand side of the first, third, and fourth rules. This variable will be bound to a set of values (built with the `?` function symbol) in the *s*-narrowing, thus allowing the program to use different values:

```
dyStep(M) -> p(M, M) .
dyStep(p(M1, M2)) -> M1 ? M2 .
dyStep(M) -> enc(M, M) .
dyStep(M) -> decrypt(M, M) .
```

Finally, we define a function `attack` that returns `true` if the secret, `ma1`, is found:

```
attack(M) -> secret(discover(M)) .
secret(ma1) -> true .
```

Once this module is loaded into the prototype, we can use *s*-narrowing to find the initial information required to break the protocol, i.e. the instantiation of `X` for this goal:

```
Maude> (narrowing attack(X) .)
{true, X -> ma1}
```

This result shows the trivial answer: if we already possess the secret information the attack is successful. We can ask for more interesting answers with the `cont` command:

```
Maude> (cont .)
{true, X -> p(ma1, V:Exp)}
```

In this case the tool deduces that we can split the pair and use the secret. Using this command we find several other possible attacks, like:

```
{true, X -> p(enc(ma1, k1), V#1:Exp) ? p(inv(k1), V#2:Exp)}
```

which indicates that we can split the pairs and use the inverse of `k1` to decrypt `ma1`. After many other results, the tool answers that the substitution `k1 ? k2` allows us to find the secret by using it in the first message sent by `alice`.

5 Concluding Remarks and Ongoing Work

In this work we propose a new narrowing relation for called *s*-narrowing that is based on the novel notion of *s*-unification, a modification of syntactic unification that allows variables to be bound to sets of expressions. It has been devised with the aim of improving the completeness results of classic narrowing. Although we think that *s*-unification has great potential, we still have to develop the theory of *s*-unification so we can use it to prove the adequacy of *s*-unification. This proposal has been implemented in a Maude prototype that allows us to study their expressivity and possible applications. The prototype uses an adaptation to *s*-narrowing of natural narrowing [9] as its on-demand strategy, thus providing an efficient implementation that allows us to use the tool with complex examples—see <http://gpd.sip.ucm.es/snarrowing> for more programs.

Regarding future work, our priority is proving the adequacy of *s*-narrowing, which implies defining an adequate order over *s*-unifiers. Besides, we should prove that our adaptation of natural narrowing to *s*-unification is still complete and optimal. We also consider expanding the prototype with search commands in the style of Maude to specify the shape of the solutions, thus avoiding irrelevant results.

References

1. Antoy, S., Hanus, M.: Overlapping Rules and Logic Variables in Functional Logic Programs. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 87–101. Springer, Heidelberg (2006)
2. Antoy, S., Hanus, M.: Functional logic programming. *Communications of the ACM* 53(4), 74–85 (2010)
3. Baader, F., Nipkow, T.: *Term Rewriting and All That*. Cambridge University Press (1998)
4. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) *Handbook of Automated Reasoning*, pp. 445–532. Elsevier and MIT Press (2001)
5. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. *Theoretical Computer Science* 236, 35–132 (2000)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Dolev, D., Yao, A.C.-C.: On the security of public key protocols. *IEEE Transactions on Information Theory* 29(2), 198–207 (1983)
8. Durán, F., Eker, S., Escobar, S., Meseguer, J., Talcott, C.L.: Variants, unification, narrowing, and symbolic reachability in maude 2.6. In: Schmidt-Schauß, M. (ed.) *RTA. LIPIcs*, vol. 10, pp. 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
9. Escobar, S.: Implementing Natural Rewriting and Narrowing Efficiently. In: Kameyama, Y., Stuckey, P.J. (eds.) *FLOPS 2004*. LNCS, vol. 2998, pp. 147–162. Springer, Heidelberg (2004)
10. Hanus, M.: *Functional logic programming: From theory to Curry*. Technical report, Christian-Albrechts-Universität Kiel (2005)
11. Hullot, J.: Canonical Forms and Unification. In: Bibel, W., Kowalski, R. (eds.) *Automated Deduction*. LNCS, vol. 87, pp. 318–334. Springer, Heidelberg (1980)
12. López-Fraguas, F.J., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: A Fully Abstract Semantics for Constructor Systems. In: Treinen, R. (ed.) *RTA 2009*. LNCS, vol. 5595, pp. 320–334. Springer, Heidelberg (2009)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1), 73–155 (1992)
14. Meseguer, J., Roşu, G.: The rewriting logic semantics project. *Theoretical Computer Science* 373(3), 213–237 (2007)
15. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation* 20(1-2), 123–160 (2007)
16. Middeldorp, A., Hamoen, E.: Completeness results for basic narrowing. *Appl. Algebra Eng. Commun. Comput.* 5, 213–253 (1994)
17. Riesco, A., Rodríguez-Hortalá, J.: Generators: Detailed proofs. Technical Report 07/12, DSIC (2012), <http://gpd.sip.ucm.es/snarrowing>
18. Riesco, A., Rodríguez-Hortalá, J.: S-narrowing for constructor systems: Detailed proofs. Technical report, DSIC (2012), <http://gpd.sip.ucm.es/snarrowing>
19. Rodríguez-Hortalá, J.: A hierarchy of semantics for non-deterministic term rewriting systems. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) *Proceedings Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008*. LIPICS, vol. 2, pp. 328–339. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2008)
20. Runciman, C., Naylor, M., Lindblad, F.: Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In: Gill, A. (ed.) *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008*, pp. 37–48. ACM (2008)
21. Thati, P., Meseguer, J.: Complete symbolic reachability analysis using back-and-forth narrowing. *Theoretical Computer Science* 366(1-2), 163–179 (2006)