Functional Logic Programming in Maude[*]

Santiago Escobar

DSIC-ELP, Universitat Politècnica de València, Spain
sescobar@dsic.upv.es

Abstract. Functional logic programming languages combine the most important features of functional programming languages and logic programming languages. Functional logic programming applied to the Maude specification language would replace the functional viewpoint by an equational viewpoint while retaining the logic features. This paper tries to bridge the gap between functional logic languages and the current implementation of narrowing as symbolic reachability in Maude. It illustrates how many features available in modern functional logic languages are easily definable and simulated in Maude but also shows how Maude goes beyond standard practices in the functional logic area by using, e.g. equational properties such as associativity and commutativity or order-sorted information. As a practical application we use the *Missionaries and Cannibals* equational logic program given by Goguen and Meseguer for Eqlog in the eighties.

1 Introduction

Functional logic programming languages combine the most important features of functional programming languages such as Haskell and logic programming languages such as Prolog. From the functional paradigm they borrow algebraic data types, advanced typing, evaluation strategies, and higher-order functions among other features; and from the logic paradigm they borrow logical variables, computing with partial information, constraint solving, and nondeterministic search for solutions among other features. Functional logic programming in the Maude specification language combines logical variables, computing with partial information, and constraint solving with reasoning modulo equational properties, advanced data types, order-sorted typing, efficient equational evaluation, distinction between concurrent and functional parts, and parameterised modules.

The Eqlog programming language [22] developed by Goguen and Meseguer in the eighties was a first attempt to combine both equational programming with logic programming. Eqlog unified equational programming and Horn-logic programming into one paradigm. Its logic design task was to embed order-sorted equational logic and Horn logic without equality into a suitable Horn logic with equality [23]. During the eighties, also the japanese *Fifth Generation Computer*

[*] This paper has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2010-21062-C02-02, and by Generalitat Valenciana PROMETEO2011/052.

System project started and it is claimed that failed because of the choice of *concurrent constraint logic programming* as the bridge between the parallel computer and the programming language, but probably it failed because most of the appropriate technology was missing. Many researchers in the functional logic programming area (see [39, 21, 11, 33, 24]) have tried, since the eighties, to combine the best features of both paradigms into a concurrent constraint functional logic programming paradigm and many possibilities have been explored (see [24, 25] for a survey). Nowadays there is a remarkable body of programming languages and tools in the functional logic area. Maude with logic features may easily be an excellent choice in the near future for an effective and efficient concurrent constraint functional logic programming language, in the spirit of the original japanese *Fifth Generation Computer System* project.

Modern concurrent constraint functional logic programming languages, such as Curry [26], combine different features of both functional and logic paradigms using an evaluation mechanism called narrowing. *Narrowing* is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern matching by unification in order to (non-deterministically) reduce these terms. Narrowing was originally introduced for automated theorem proving [41], then used as a mechanism for solving equational unification problems [20], it became the "de facto" evaluation mechanism for functional logic programming languages [5], and it was generalized from equational unification problems to solve the more general problem of symbolic reachability [35]. The narrowing mechanism has a number of important applications including automated proofs of termination [7], execution of functional-logic programming languages [6], program transformation [1], verification of cryptographic protocols [35], and equational unification [28], to mention just a few.

An essential aspect in concurrent constraint functional logic programming is the choice of an effective and efficient narrowing evaluation strategy. Several approaches have been defined in the literature [5, 2, 4, 15, 16, 18, 12]. The *needed narrowing* strategy [5] and the *parallel needed narrowing* strategy [4], both extended with the *residuation* principle, are applicable to left-linear constructor rewrite systems and are lazy (or demand-driven), obtaining interesting properties and performance. The *natural narrowing* strategy [15, 16] is applicable to left-linear constructor rewrite systems too, and is extended to rewrite systems in general [18]. It is also demand-driven with similar interesting properties and performance. The development of narrowing in Maude as symbolic reachability [9, 12] is applicable to any unconditional rewrite theory without memberships (up to some extra conditions [35, 9, 12]). This version of narrowing in `Maude` is not demand-driven and it is still an open problem to develop demand-driven narrowing evaluation strategies dealing with equational properties such as associativity and commutativity.

This paper tries to bridge the gap between functional logic languages such as Curry and the current implementation of narrowing as symbolic reachability in Maude. It illustrates how many features available in modern functional logic languages are easily definable and simulated in Maude; we consider: (i) a semantics of values, (ii) a call-time choice semantics, (iii) conditional rules, (iv) strict equality, (v) extra variables, (vi) constraint solving, and (vii) residuation. Many other features of functional logic languages are not considered in this paper because of lack of space. However, this paper shows how Maude goes beyond standard practices in the functional logic area by using features not available to functional logic languages, e.g. reasoning modulo and an order-sorted setting.

As a motivating example for the reader, in Section 2 we present how the *Missionaries and Cannibals* equational logic program of [22] can be written using the narrowing features available nowadays in Maude. This is an example requiring some constraint solving features, logical variables, order-sorted types, and associativity–commutativity–identity, thus it cannot be specified in current functional logic languages. In Section 3 we introduce some basic concepts on rewriting logic. In Section 4 we recall the narrowing mechanism and how it is made available in Maude. In Section 5 we present how features available in Curry are easily definable and simulated in Maude and demonstrate in Section 6 how queries on the motivating example are executed. Finally, we conclude in Section 7.

2 Example: Missionaries and Cannibals

As a motivating example for the reader, we present how the *Missionaries and Cannibals* equational logic program of [22] can be written using the narrowing features currently available in Maude. The equational logic program[1] of [22] used a syntax proposed for Eqlog where a functional syntax very close to Maude was combined with some syntax for Horn-clauses using symbol ":-".

```
module MAC[T :: MACTH] using NAT, TRIPLIST = LIST[trip] is
  preds
    boatok : trip
    solve, good : triplist
  fns
    boat : pred -> trip
    lb,rb : triplist -> pset
    mset,cset : pset -> pset
  vars
    PS:pset, L:triplist, P:person, T:trip
  axioms
    boatok(boat(PS)) :- # PS = 1.
    boatok(boat(PS)) :- # PS = 2.
    mset(PS) = PS /\ m0.
    cset(PS) = PS /\ c0.
    lb(nil) = m0 + c0.
    rb(nil) = empty.
    lb(L * boat(PS)) = lb(L) - PS :- even # L.
    rb(L * boat(PS)) = rb(L) + PS :- even # L.
    rb(L * boat(PS)) = rbQ - PS :- odd # L.
    lb(L * boat(PS)) = lb(L) + PS :- odd # L.
    good(L * T) :- # cset(lb(L * T)) =< # mset(lb(L * T)) or mset(lb(L * T)) = 0,
```

[1] The original program in [22] had an error because the number of cannibals has to be lower than or equal to the number of missionaries in both sides unless there is no missionary. We discovered this error thanks to the new narrowing-based executability in Maude.

```
                # cset(rb(L * T)) =< # mset(rb(L * T)) or mset(rb(L * T)) = 0,
                good(L), boatok(T).
   good(nil).
   solve(L) :- good(L), lb(L) = empty.
endmod MAC
```

This module is parametric on a theory `T :: MATCH` for the names of the missionaries and cannibals, which are instantiated to `m0 = taylor, helen, william` and `c0 = umugu, nzwave, amoc`. Also a module for lists is imported[2], where `_*_` is the constructor symbol for lists and `#_` is the length operation for lists. The system is configured as a list of trips (sort `triplist`) where each trip is a term rooted by a predicate `boat` with a set of names of missionaries and cannibals. Each trip in the list is considered `good` if it satisfies some properties. Odd positions in the list represent moving from left to right and even positions from right to left. There are some extra symbols for set manipulation: `_+_` for union, `_-_` for removal, and `_/\_` for intersection; indeed (multi-)sets are the only data structure in this example with extra equational properties, namely associativity, commutativity and identity for the multiset. There are also some symbols for lists: `even` indicates whether a list has an even number of elements and `odd` indicates whether a list has an odd number of elements. Finally, the predicate `boatok` checks whether a trip is ok and `solve` is the general predicate for checking/generating the `triplist` solution.

This is an example requiring some constraint solving features, logical variables, order-sorted types, and associativity, commutativity, and an identity symbol for multisets. For instance, it requires constraint solving features because of the numerical conditions for length of lists in the conditions of predicate `good`; this would be solved by using a generator function and using these length functions by residuation. Also, the program considers equational properties, since the problem is represented by a list of trips and each element is the boat with a multiset of missionaries and cannibals; this would be easily handled by narrowing modulo these properties. And it clearly includes order-sorted information in the sense of having people which are specialized into missionaries and cannibals. Also, note that some functional logic features described in Section 5 are necessary here; for instance, this example considers a semantics of values instead of all reachable terms, predicates are just conditional rules evaluated into a special sort different from `Bool` that only contains positive (or successful) cases, and conditions in conditional rules are indeed strict equalities instead of syntactic equality.

3 Background on Rewriting Logic and Term Rewriting

We follow the classical notation and terminology from [42] for term rewriting, and from [32] for rewriting logic and order-sorted notions. We assume an order-sorted signature $\mathbf{\Sigma} = (\mathsf{S}, \leq, \Sigma)$ with poset of sorts $(\mathsf{S}, \leq)$ and such that for each

[2] The original program assumes lists are created using an associative symbol but unification modulo associativity is infinitary and it is not available in Maude.

sort $\mathsf{s} \in \mathsf{S}$ the connected component of s in $(\mathsf{S}, \leq)$ has a top sort, denoted $[\mathsf{s}]$, and all $f : \mathsf{s}_1 \cdots \mathsf{s}_n \to \mathsf{s}$ with $n \geq 1$ have a top sort overloading $f : [\mathsf{s}_1] \cdots [\mathsf{s}_n] \to [\mathsf{s}]$. We also assume an S-sorted family $\mathcal{X} = \{\mathcal{X}_\mathsf{s}\}_{\mathsf{s} \in \mathsf{S}}$ of disjoint variable sets with each $\mathcal{X}_\mathsf{s}$ countably infinite. $\mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ is the set of terms of sort s, and $\mathcal{T}_{\Sigma,\mathsf{s}}$ is the set of ground terms of sort s. We write $\mathcal{T}_\Sigma(\mathcal{X})$ and $\mathcal{T}_\Sigma$ for the corresponding order-sorted term algebras. For a term t, $\mathit{Var}(t)$ denotes the set of all variables in t.

Positions are represented by sequences of natural numbers denoting an access path in the term when viewed as a tree. The top or root position is denoted by the empty sequence Λ. We define the relation $p \leq q$ between positions as $p \leq p$ for any p; and $p \leq p.q$ for any p and q. Given $U \subseteq \Sigma \cup \mathcal{X}$, $\mathit{Pos}_U(t)$ denotes the set of positions of a term t that are rooted by symbols or variables in U. The set of positions of a term t is written $\mathit{Pos}(t)$, and the set of non-variable positions $\mathit{Pos}_\Sigma(t)$. The subterm of t at position p is $t|_p$ and $t[u]_p$ is the term t where $t|_p$ is replaced by u.

A *substitution* $\sigma \in \mathcal{S}\mathit{ubst}(\Sigma, \mathcal{X})$ is a sorted mapping from a finite subset of $\mathcal{X}$ to $\mathcal{T}_\Sigma(\mathcal{X})$. Substitutions are written as $\sigma = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ where the domain of σ is $\mathit{Dom}(\sigma) = \{X_1, \ldots, X_n\}$ and the set of variables introduced by terms $t_1, \ldots, t_n$ is written $\mathit{Ran}(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}_\Sigma(\mathcal{X})$. The application of a substitution σ to a term t is denoted by $t\sigma$. For simplicity, we assume that every substitution is idempotent, i.e., σ satisfies $\mathit{Dom}(\sigma) \cap \mathit{Ran}(\sigma) = \emptyset$. Substitution idempotency ensures $t\sigma = (t\sigma)\sigma$. The restriction of σ to a set of variables V is $\sigma|_V$; sometimes we write $\sigma|_{t_1,\ldots,t_n}$ to denote $\sigma|_V$ where $V = \mathit{Var}(t_1) \cup \cdots \cup \mathit{Var}(t_n)$. Composition of two substitutions σ and σ' is denoted by $\sigma\sigma'$.

A *Σ-equation* is an unoriented pair $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ for some sort $\mathsf{s} \in \mathsf{S}$. Given an order-sorted signature Σ and a set $\mathcal{E}$ of Σ-equations, order-sorted equational logic induces a congruence relation $=_\mathcal{E}$ on terms $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$ (see [34]). The $\mathcal{E}$-equivalence class of a term t is denoted by $[t]_\mathcal{E}$ and $\mathcal{T}_{\Sigma/\mathcal{E}}(\mathcal{X})$ and $\mathcal{T}_{\Sigma/\mathcal{E}}$ denote the corresponding order-sorted term algebras modulo $\mathcal{E}$. Throughout this paper we assume that $\mathcal{T}_{\Sigma,\mathsf{s}} \neq \emptyset$ for every sort s, because this affords a simpler deduction system.

An *equational theory* $(\Sigma, \mathcal{E})$ is a pair with Σ an order-sorted signature and $\mathcal{E}$ a set of Σ-equations. The *$\mathcal{E}$-subsumption* preorder $\sqsupseteq_\mathcal{E}$ (or just $\sqsupseteq$ if $\mathcal{E}$ is understood) holds between $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})$, denoted $t \sqsupseteq_\mathcal{E} t'$ (meaning that t is *more general* than t' modulo $\mathcal{E}$), if there is a substitution σ such that $t\sigma =_\mathcal{E} t'$; such a substitution σ is said to be an *$\mathcal{E}$-match* from t to t'.

An *$\mathcal{E}$-unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_\mathcal{E} t'\sigma$. For $\mathit{Var}(t) \cup \mathit{Var}(t') \subseteq W$, a set of substitutions $\mathit{CSU}^W_\mathcal{E}(t = t')$ is said to be a *complete* set of unifiers for the equality $t = t'$ modulo $\mathcal{E}$ away from W iff: (i) each $\sigma \in \mathit{CSU}^W_\mathcal{E}(t = t')$ is an $\mathcal{E}$-unifier of $t = t'$; (ii) for any $\mathcal{E}$-unifier ρ of $t = t'$ there is a $\sigma \in \mathit{CSU}^W_\mathcal{E}(t = t')$ such that $\sigma|_W \sqsupseteq_\mathcal{E} \rho|_W$; (iii) for all $\sigma \in \mathit{CSU}^W_\mathcal{E}(t = t')$, $\mathit{Dom}(\sigma) \subseteq (\mathit{Var}(t) \cup \mathit{Var}(t'))$ and $\mathit{Ran}(\sigma) \cap W = \emptyset$. If the set of variables W is irrelevant or is understood from the context, we write $\mathit{CSU}_\mathcal{E}(t = t')$ instead of $\mathit{CSU}^W_\mathcal{E}(t = t')$. An $\mathcal{E}$-unification algorithm is *complete* if for any equation

$t = t'$ it generates a complete set of $\mathcal{E}$-unifiers. A unification algorithm is said to be *finitary* and complete if it always terminates after generating a finite and complete set of solutions.

A *rewrite rule* is an oriented pair $l \to r$, where[3] $l \notin \mathcal{X}$ and $l, r \in \mathcal{T}_{\Sigma}(\mathcal{X})_{\mathsf{s}}$ for some sort $\mathsf{s} \in \mathsf{S}$. An *(unconditional) order-sorted rewrite theory* is a triple $(\Sigma, \mathcal{E}, R)$ with Σ an order-sorted signature, $\mathcal{E}$ a set of Σ-equations, and R a set of rewrite rules.

The rewriting relation on $\mathcal{T}_{\Sigma}(\mathcal{X})$, written $t \to_R t'$ or $t \to_{p,R} t'$ holds between t and t' iff there exist $p \in Pos_{\Sigma}(t)$, $l \to r \in R$ and a substitution σ, such that $t|_p = l\sigma$, and $t' = t[r\sigma]_p$. The relation $\to_{R/\mathcal{E}}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is $=_{\mathcal{E}};\to_R;=_{\mathcal{E}}$, i.e., $t \to_{R/\mathcal{E}} t'$ iff there exists u, u' s.t. $t =_{\mathcal{E}} u \to_R u' =_{\mathcal{E}} t'$. Note that $\to_{R/\mathcal{E}}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ induces a relation $\to_{R/\mathcal{E}}$ on the free $(\Sigma, \mathcal{E})$-algebra $\mathcal{T}_{\Sigma/\mathcal{E}}(\mathcal{X})$ by $[t]_{\mathcal{E}} \to_{R/\mathcal{E}} [t']_{\mathcal{E}}$ iff $t \to_{R/\mathcal{E}} t'$. The transitive (resp. transitive and reflexive) closure of $\to_{R/\mathcal{E}}$ is denoted $\to^{+}_{R/\mathcal{E}}$ (resp. $\to^{*}_{R/\mathcal{E}}$).

The application of one $\to_{R/\mathcal{E}}$ step is undecidable in general since $\mathcal{E}$-congruence classes can be arbitrarily large. Therefore, $R/\mathcal{E}$-rewriting is usually implemented [29] by R,$\mathcal{E}$-rewriting. A relation $\to_{R,\mathcal{E}}$ on $\mathcal{T}_{\Sigma}(\mathcal{X})$ is defined as: $t \to_{p,R,\mathcal{E}} t'$ (or just $t \to_{R,\mathcal{E}} t'$) iff there exist $p \in Pos_{\Sigma}(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_{\mathcal{E}} l\sigma$ and $t' = t[r\sigma]_p$.

We assume that the relation $\to_{R,\mathcal{E}}$ is local $\mathcal{E}$-*coherent* [29], i.e., $\forall t_1, t_2, t_3$ we have $t_1 \to_{R,\mathcal{E}} t_2$ and $t_1 =_{\mathcal{E}} t_3$ implies $\exists t_4, t_5$ such that $t_2 \to^{*}_{R,\mathcal{E}} t_4$, $t_3 \to^{+}_{R,\mathcal{E}} t_5$, and $t_4 =_{\mathcal{E}} t_5$. Let us recall how coherence works at least for the common associative-commutative (AC) case. The best way to illustrate it is by its *absence*. Consider a symbol _+_ declared as AC. Now consider the rule $b + b \to c$, where b and c are constants. This rule, if not completed by another, is *not* coherent modulo AC. What this means is that there will be term *contexts* in which the rule *should* be applied, but it cannot be applied. Consider, for example, the term $b + (a + b)$, where a is also a constant. Intuitively, we should be able to apply to it the above rule to simplify it to the term $a + c$ in one step. However, since we are using the weaker rewrite relation $\to_{R,AC}$ instead of the stronger but much harder to implement relation $\to_{R/AC}$, we cannot! The problem is that the rule cannot be applied (even if we match modulo AC) to either the top term $b + (a + b)$ or the subterm $a + b$. We can however make our rule *coherent* modulo *AC* by adding the extra rule $b + b + Y \to c + Y$. This extended version of our rule will now apply to the term $b + (a + b)$, giving the simplification $b + (a + b) \longrightarrow_{R,AC} a + c$. Technically, what coherence means is that the weaker relation $\to_{R,\mathcal{E}}$ becomes semantically equivalent to the stronger relation $\to_{R/\mathcal{E}}$.

Coherence can be handled implicitly or explicitly, i.e., either the matching mechanism is modified to take care of this issue or the rules are explicitly ex-

[3] Note that we do not impose here the standard condition $Var(r) \subseteq Var(l)$, necessary for executability of rewriting in practice. Rewriting with extra variables in right-hand sides is handled at a theoretical level by allowing the matching substitution to instantiate these extra variables in any possible way. Extra variables do no pose any problem to narrowing and are part of the nondeterministic search of solutions typical of logic programming.

tended, which is the option shown above; see [43] for a comparison between implicit and explicit extensions. For rewriting, implicit extensions are sufficient in many cases, as the implicit $\mathcal{E}$-coherence completion provided by the Maude tool [10] for any combination of associativity (A), commutativity (C), and identity (U) axioms. For narrowing, implicit extension is more complicated and it is sufficient to consider explicit single-variable extensions in common cases such as combinations of C, AC, and ACU axioms, i.e., given a rule $s \to t$ one considers $s + x \to t + x$ where x is a new variable. The method is as follows for AC. For any symbol f which is AC, and for any rule of the form $f(u, v) \to w$ in $\mathcal{E}$, we add also the equation $f(f(u, v), X) \to f(w, X)$, where X is a new variable not appearing in u, v, w. In an order-sorted setting, we should give to X *the biggest sort possible*, so that it will apply in all generality. As an additional optimization, note that some rules may already be coherent modulo AC, so that we need not add the extra equation. See [13] for further information.

We also assume that the equational theory is split into $\mathcal{E} = E \cup Ax$ such that E is a set of equations oriented into rules and Ax is a set of equational axioms satisfying:

1. Ax is *regular*, i.e., for each $t = t'$ in Ax, we have $Var(t) = Var(t')$, and *sort-preserving*, i.e., for each substitution σ, we have $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ iff $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$; furthermore, for each equation $t = t'$ in Ax, all variables in $Var(t)$ have a top sort.
2. Ax has a finitary and complete unification algorithm, which implies that Ax-matching is finitary and complete.
3. For each $t \to t'$ in E we have $Var(t') \subseteq Var(t)$.
4. E is *sort-decreasing*, i.e., for each $t \to t'$ in E, each $\mathsf{s} \in \mathsf{S}$, and each substitution σ, $t'\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$ implies $t\sigma \in \mathcal{T}_\Sigma(\mathcal{X})_\mathsf{s}$.
5. The relation $\to_{E,Ax}$ is confluent, terminating, and local Ax*-coherent*, i.e., for each term t, the relation terminates and produces a unique irreducible term (up to Ax-equivalence) denoted by $t{\downarrow}_{E,Ax}$.

Given an order-sorted equational theory $(\Sigma, E \cup Ax)$, (t', θ) is an E, Ax-*variant* [19] (or just a variant) of term t if $t\theta{\downarrow}_{E,Ax} =_{Ax} t'$ and $\theta{\downarrow}_{E,Ax} =_{Ax} \theta$. An order-sorted equational theory $(\Sigma, E \cup Ax)$ has the *finite variant property* [19] iff for each Σ-term t, a complete set of its most general variants is finite. A finitary and complete unification algorithm is defined for order-sorted equational theories with the finite variant property [19].

4 Narrowing in Maude

Logic programming languages are well suited for goal solving. Functional programming languages are equipped with equational definition of operations. Several approaches have been considered in the literature for combining the funcional and logic paradigms, see [24]. On the one hand, it is a natural idea to add an equality predicate to logic programs, leading to equational logic programming [27]. On the other hand, it is also a natural idea to add logical variables to functional programs, leading to narrowing-based equational reasoning [20]. Logic variables are also valuable at the level of model checking rather than functional programming, as proposed for symbolic reachability in [35] and extended to *logical* model checking in [17, 8].

At each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered. Similarly, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation[4] on the variables of the subject term and the rule's left-hand side are going to be considered. The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \to r$ to rewrite t at a position p in t, but narrowing unifies the left-hand side l and the chosen subject term $t|_p$ before actually performing the rewriting step. Narrowing is restricted[5] to non-variable positions of t, whereas rewriting does not require such a restriction.

Let $\mathcal{R} = (\Sigma, E \cup Ax, R)$ be an order-sorted rewrite theory where R is a set of unconditional rewrite rules, specified with the `rl` keyword, E is a set of unconditional equations specified with the `eq` and `variant` keywords, and Ax is a set of commonly occurring axioms —declared in Maude as equational attributes— such that an $E \cup Ax$-unification procedure is available in Maude. Unification algorithms already available in Maude are divided in two groups: (i) Ax-unification for order-sorted signatures with any combination of free, C, AC, or ACU function symbols [9], and (ii) $E \cup Ax$-unification for order-sorted equational theories with the finite variant property [12].

Let $CSU_{E\cup Ax}(u = u')$ provide[6] a finitary and complete set of unifiers for any pair of terms u, u' with the same top sort. The R,$(E \cup Ax)$-*narrowing* relation on $\mathcal{T}_\Sigma(\mathcal{X})$ is defined as $t \leadsto_{\sigma,p,R,E\cup Ax} t'$ (or $\leadsto_\sigma$ when p, R, E, Ax are understood) if there is a non-variable position $p \in Pos_\Sigma(t)$, a (possibly renamed) rule $l \to r$ in R, and a unifier $\sigma \in CSU_{E\cup Ax}(t|_p = l)$ such that $t' = (t[r]_p)\sigma$. We denote by $t \leadsto^{+}_{\sigma,R,E\cup Ax} t'$ (resp. $t \leadsto^{*}_{\sigma,R,E\cup Ax} t'$) the transitive (resp. reflexive-transitive) closure of the narrowing relation, where σ is obtained as the composition of the substitutions for each narrowing step in the sequence.

Consider the following system module defining the addition function `_+_` on natural numbers built from `0` and `s`:

```
mod NAT-NARROWING is
  sort Nat .
```

[4] Demand-driven narrowing strategies may require instantiations of a term that do not correspond to a most general unifier of a subterm and a left-hand side of a rule, see [5, 2].

[5] The *paramodulation* inference rule used in paramodulation-based theorem proving [37] is similar to narrowing and does not require non-variable positions.

[6] In the present implementation of Maude, we are not interested in a minimal set of unifiers, but only in a finite and complete set. Minimality is easily achieved in syntactic unification (see [31]) but it is very costly in Ax-unification or $E \cup Ax$-unification, e.g., the ACU-unification available in Maude does not always provide a minimal set of unifiers.

```
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
    op _+_ : Nat Nat -> Nat .
    vars X Y : Nat .
    rl [base] : 0 + Y => Y .
    rl [ind] : s(X) + Y => s(X + Y) .
  endm
```

Consider the term `X + s(0)` and the two rules `base` and `ind`. Narrowing will instantiate variable `X` with `0` and `s(X')` respectively in order to be able to apply each of these rules, i.e., the following two narrowing steps are generated:

$$\texttt{X} + \texttt{s(0)} \leadsto_{\{\texttt{X}\mapsto\texttt{0}\},\texttt{base}} \texttt{s(0)}$$

$$\texttt{X} + \texttt{s(0)} \leadsto_{\{\texttt{X}\mapsto\texttt{s(\#1:Nat)}\},\texttt{ind}} \texttt{s(\#1:Nat} + \texttt{s(0))}$$

Note that, for simplicity, we show only the bindings of the unifier that affect the input term. There are infinitely many narrowing derivations starting at the input expression `X + s(0)` (at each step the reduced subterm is underlined):

1. $\underline{\texttt{X} + \texttt{s(0)}} \leadsto_{\{\texttt{X}\mapsto\texttt{0}\},\texttt{base}} \texttt{s(0)}$
2. $\underline{\texttt{X} + \texttt{s(0)}} \leadsto_{\{\texttt{X}\mapsto\texttt{s(\#1:Nat)}\},\texttt{ind}} \texttt{s(}\underline{\texttt{\#1:Nat} + \texttt{s(0)}}\texttt{)} \leadsto_{\{\texttt{\#1:Nat}\mapsto\texttt{0}\},\texttt{base}} \texttt{s(s(0))}$
3. $\underline{\texttt{X} + \texttt{s(0)}} \leadsto_{\{\texttt{X}\mapsto\texttt{s(\#1:Nat)}\},\texttt{ind}} \texttt{s(}\underline{\texttt{\#1:Nat} + \texttt{s(0)}}\texttt{)}$
   $\leadsto_{\{\texttt{\#1:Nat}\mapsto\texttt{s(\#2:Nat)}\},\texttt{ind}} \texttt{s(s(}\underline{\texttt{\#2:Nat} + \texttt{s(0)}}\texttt{))} \leadsto_{\{\texttt{\#2:Nat}\mapsto\texttt{0}\},\texttt{base}} \texttt{s(s(s(0)))}$

And some of those infinitely many narrowing derivations are infinite in length, e.g. by applying rule `ind` infinitely many times:

$$\begin{aligned}\underline{\texttt{X} + \texttt{s(0)}} &\leadsto_{\{\texttt{X}\mapsto\texttt{s(\#1:Nat)}\},\texttt{ind}} \texttt{s(}\underline{\texttt{\#1:Nat} + \texttt{s(0)}}\texttt{)}\\ &\leadsto_{\{\texttt{\#1:Nat}\mapsto\texttt{s(\#2:Nat)}\},\texttt{ind}} \texttt{s(s(}\underline{\texttt{\#2:Nat} + \texttt{s(0)}}\texttt{))}\\ &\leadsto_{\{\texttt{\#2:Nat}\mapsto\texttt{s(\#3:Nat)}\},\texttt{ind}} \texttt{s(s(s(}\underline{\texttt{\#3:Nat} + \texttt{s(0)}}\texttt{)))}\\ &\ldots\end{aligned}$$

The classical application of narrowing modulo an equational theory is to perform $E \cup Ax$-*unification* by E, Ax-narrowing (see [20, 19]) when the equations E are oriented into rules and are confluent, terminating and coherent modulo Ax (see Section 3). When the theory also satisfies the *finite variant property* [19], a finitary and complete unification algorithm based on a narrowing strategy called *folding variant narrowing* is provided in [19]. This unification algorithm is available in Maude, see [12].

The modern application of narrowing modulo an equational theory is that of *symbolic reachability analysis* [35], when the rules R are understood as *transition rules*. Given an order-sorted rewrite theory of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$ where: (i) $E \cup Ax$ has a finitary and complete $E \cup Ax$-unification algorithm (for instance, the ACU-unification algorithm available in Maude or the $E \cup Ax$-unification algorithm based on folding variant narrowing) and (ii) the transition rules R are $E \cup Ax$-coherent and *topmost* (see [35]), then narrowing is a *complete* deductive method for symbolic reachability analysis, i.e., for solving existential queries of the form $\exists \overline{x} : t(\overline{x}) \to^{*} t'(\overline{x})$ in the sense that the formula holds for $\mathcal{R}$ iff there is a sequence of narrowing steps $t \rightsquigarrow_{\sigma_1,R,E\cup Ax} t_1 \rightsquigarrow_{\sigma_2,R,E\cup Ax} t_2 \cdots t_{n-1} \rightsquigarrow_{\sigma_n,R,E\cup Ax} t_n$ such that t_n and t' have a $E\cup Ax$-unifier. This symbolic reachability is available also in Maude, see [12]. The standard search command of Maude uses the syntax `search` Term_1 *arrow* Term_2 where the arrows can be `=>1`, `=>+`, `=>*`, `=>!` for just one rewriting step, one or more rewriting steps, zero or more rewriting steps, or until no more rewriting steps are possible. This feature is extended to narrowing in Full Maude by allowing variables both in Term_1 and Term_2 (possibly sharing variables) and allowing extra arrows `~>1`, `~>+`, `~>*`, `~>!` for just one narrowing step, one or more narrowing steps, zero or more narrowing steps, or until no more narrowing steps are possible.

The current use of narrowing and unification in Maude is distributed as follows: (i) Ax-unification available in Maude for order-sorted signatures with any combination of free, C, AC, or ACU function symbols (see [9]); (ii) $E \cup Ax$-unification available in Full Maude (and soon in Maude) using the folding variant narrowing strategy for theories with the finite variant property (see [12]); and (iii) narrowing-based reachability analysis using rules R modulo $E \cup Ax$ (see [12]).

The narrowing relation currently available in Maude is slightly different than the standard one formally defined above. Let $\mathcal{R} = (\Sigma, G\cup E\cup Ax, R)$ be an order-sorted rewrite theory where R, E, and Ax are defined as above and G are the remaining equations. Note that equations in G do not have the `variant` attribute and have no restriction, i.e., they can be conditional equations, with the `owise` attribute, etc. Each narrowing step of the form $t \rightsquigarrow_{\sigma,p,R,E\cup Ax} t'$ is followed by simplification $t'{\downarrow_{G,Ax}}$, i.e., the combined relation is defined as $t \rightsquigarrow_{\sigma,p,R,E,G,Ax} t''$ iff $t \rightsquigarrow_{\sigma,p,R,E\cup Ax} t'$ and $t'' = t'{\downarrow_{G,Ax}}$. Note that this combined relation may be incomplete because equations G are not considered for unification, i.e., given a reachability problem of the form $\exists \overline{x} : t(\overline{x}) \to^{*} t'(\overline{x})$ and a solution σ (i.e., $t\sigma \to^{*}_{R,G\cup E\cup Ax} t'\sigma$), the relation $\rightsquigarrow_{\sigma,p,R,E,G,Ax}$ may not be able to find a more general solution.

5 Functional Logic Programming in Maude

We define a functional logic program in Maude as a rewrite theory $\mathcal{R} = (\Sigma, G \cup E\cup Ax, R)$ where R defines[7] the rules used by narrowing, $E\cup Ax$ is the equational theory for unification purposes and $G \cup Ax$ is the equational theory for simplification only. A functional logic computation consists of a reachability problem of the form $\exists \overline{x} : t(\overline{x}) \to^{*} t'(\overline{x})$ and a narrowing sequence with a computed substitution σ where σ is a solution. Note that we are interested in a semantics[8]

[7] In reality, one would expect two sets R_{rew} and R_{narr}, one for rewriting only and one for narrowing only, as in the equational case with E and G, but we leave this for future implementations.

[8] A well-versed reader may believe we are interested in a semantics of both computed answers and normal forms instead of only normal forms but this is arguable, e.g. the different solutions found by narrowing for the reachability problem $f(x) \to^{*} 0$ using

of normal forms and assume that the term $t'(\overline{x})$ is *strongly irreducible w.r.t.* $\rightarrow_{R,G\cup E\cup Ax}$, i.e., for any irreducible substitution $\rho : \mathcal{T}_{\Sigma}(\mathcal{X}) \rightarrow \mathcal{T}_{\Sigma}$, $t'\rho$ is irreducible. Indeed, we are interested in a semantics of values rather than normal forms but the concept of a value in Maude is not just as simple as a constructor term in Haskell or Curry. As an example, take the rule `double(X)` $\rightarrow$ `X + X` using the built-in Maude addition operator on natural numbers. For the reachability problem `double(1)` $\rightarrow^*$ `X`, the expression `double(1)` is just evaluated to `2` and assigned to `X`. If we take the reachability problem `double(1+2)` $\rightarrow^*$ `Y`, there are different evaluation orders depending on whether `1+2` is evaluated before the symbol `double` or not, but the normal form assigned to `Y` is `6`.

In this section, we consider several features that are common in functional logic languages such as Curry. In Section 5.1 we consider a call-time choice semantics for computing values. Then equality in this semantics is adapted to the notion of *strict equality* in Section 5.2 and this allows us to consider conditional rules with conditions using only strict equality in Section 5.3 and extra variables in right-hand sides of rules in Section 5.4. Finally, all these features provide us with all the ingredients for constraint solving capabilities in Section 5.5, including the concept of residuation.

5.1 Non-deterministic functional logic computations and kinds in order-sorted equational logic

An interesting property of functional logic programming languages is that they do not assume confluence of the equational specification. For instance, consider the non-deterministic function `coin` with two rules `coin` $\rightarrow 0$ and `coin` $\rightarrow 1$. When we consider the expression `double(coin)`, the obtained results are `0`, `1`, and `2` even if the reader would expect only `0` and `2`. Different semantics give different results to the previous expression:

1. If the expression `coin` is passed without evaluation to the function `double`, we obtain the expression `coin+coin`, which has four possible derivations to values `0`, `1`, `1`, and `2`. This behaviour corresponds to *run-time choice*, which means that the choice of the value associated to a function parameter may be determined later. This is the standard semantics associated to rewrite theories in Maude.
2. If the expression `coin` is evaluated before passing it to the function `double`, we obtain the expressions `0+0` and `1+1`, which return `0` and `2`. This behaviour corresponds to *call-time choice* in functional/equational programming, which means that the choice of the value associated to a function parameter is determined when calling the function symbol. This behaviour corresponds also to *variable sharing* when non-determinism is present. Sharing in rewriting and narrowing are captured by the idea of graph rewriting [38] and graph narrowing [14]. This is the intended semantics accepted by the functional

programs (i) $f(x) \rightarrow 0$ and $f(0) \rightarrow 0$ and (ii) $f(x) \rightarrow 0$ are irrelevant for the folding variant narrowing strategy [19], which returns *only* the most general solution using the common rule $f(x) \rightarrow 0$, and similarly for this paper.

logic community. See [36] for elaborated interactions on non-determinism and non-right-linear equations (or rules). Some bizarre behaviours are still possible when combining it with demand-driven evaluation and solutions have been recently developed [40].

All these behaviours are easily representable in rewriting logic by using kinds. Since functional logic programs are interested only in values and not in normal forms, constructor symbols would be the only ones belonging to a concrete sort and defined symbols will belong to the kind. For example, let us consider a definition of natural numbers using the sort `Nat`, without any algebraic property; where the kind of `Nat`, `[Nat]`, is used for terms that are not natural numbers.

```
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
```

Operations, for example addition and the function `double` would be defined on the kind of `Nat`, since they are not considered as valid terms of sort `Nat`.

```
  op _+_ : [Nat] [Nat] -> [Nat] .
  rl 0 + M:[Nat] => M:[Nat] .
  rl s N:[Nat] + M:[Nat] => s (N:[Nat] + M:[Nat]) .

  op double : [Nat] -> [Nat] .
  rl double(N:[Nat]) => N:[Nat] + N:[Nat] .
```

Similarly, the function `coin` is defined on the kind.

```
  op coin : -> [Nat] .
  rl coin => 0 .
  rl coin => s 0 .
```

Now we can search for all possible normal forms, corresponding to a typical run-time choice semantics.

```
search double(coin) =>! N:[Nat] .
Solution 1          Solution 2         Solution 3
N:[Nat] --> 0       N:[Nat] --> s 0    N:[Nat] --> s s 0
No more solutions.
```

We can force a call-time choice semantics by imposing variables and constructor symbols of sort `Nat` instead of the kind `[Nat]`, which forces arguments to be evaluated first.

```
op double : Nat -> [Nat] .               search double(coin) =>! N:[Nat] .
rl double(N:Nat) => N:Nat + N:Nat .      Solution 1       Solution 2
                                         N:[Nat] --> 0    N:[Nat] --> s s 0
                                         No more solutions.
```

As a typical example of non-determinism in functional logic programming with a call-time choice, we include the function `permute` based on a function `insert` that non-deterministically inserts an element in any position of a listof natural numbers.

```
mod PERMUTE is
  sort Nat .                        sort NatList .
  op 0 : -> Nat [ctor] .            op nil : -> NatList [ctor] .
  op s_ : Nat -> Nat [ctor] .       op _:_ : Nat NatList -> NatList [ctor] .
  vars N E : Nat . var NL : NatList .

  op permute : NatList -> [NatList] .
  rl permute(nil) => nil .
  rl permute(N : NL) => insert(N,permute(NL)) .

  op insert : Nat NatList -> [NatList] .
  rl insert(E,nil) => E : nil .
  rl insert(E,N : NL) => E : N : NL .
  rl insert(E,N : NL) => N : insert(E,NL) .
endm
```

A typical evaluation would return all the permutations of a given list of natural numbers.

```
search permute(0 : s 0 : s s 0 : nil) =>! NL:NatList .
Solution 1                          Solution 2
NL:NatList --> 0 : s 0 : s s 0 : nil NL:NatList --> s 0 : 0 : s s 0 : nil
Solution 3                          Solution 4
NL:NatList --> 0 : s s 0 : s 0 : nil NL:NatList --> s 0 : s s 0 : 0 : nil
Solution 5                          Solution 6
NL:NatList --> s s 0 : 0 : s 0 : nil NL:NatList --> s s 0 : s 0 : 0 : nil
No more solutions.
```

We can already use the narrowing capabilities to solve some symbolic reachability problems, for instance give a fully instantiated final term and include logical variables in the initial term, so that narrowing searches for solutions; note that the inclusion of the variable `Z:NatList` makes the search space infinite even if there is only one solution, which is obtained by restricting the search to the first solution found using the extra option `[1]`.

```
search [1] permute(0 : X:Nat : (s s 0) : Z:NatList)
       ~>! 0 : (s 0) : (s s 0) : nil .
Solution 1
X:Nat --> s 0 ; Z:NatList --> nil
No more solutions.
```

5.2 Strict equality

Since functional logic programs are interested on values instead of normal forms, the standard Maude equality is not useful and we need a *strict equality*[9] [26]. That is, an expression `coin == 0` will always be evaluated to false because it requires a rule application of `coin` before checking for equality. We would like to have a built-in strict equality, as in major functional logic languages, but we have to define it explicitly in this paper. Indeed, since we want narrowing to be able to instantiate variables in a proper way, we must implement an explicit strict equality in every program written in Maude. We use the symbol `=:=` along[10] the paper to denote this strict equality, which is explicitly defined for each sort in the following form, when there are no algebraic properties, using also the boolean symbol `and`:

$$\begin{array}{rcll} f(X_1,\ldots,X_n) \texttt{ =:= } g(Y_1,\ldots,Y_m) & \to & false & \text{if } f \neq g, n \geq 0, m \geq 0 \\ c \texttt{ =:= } c & \to & true & \\ f(X_1,\ldots,X_n) \texttt{ =:= } f(Y_1,\ldots,Y_n) & \to & X_1\texttt{=:=}Y_1 \texttt{ and } \cdots \texttt{ and } X_n\texttt{=:=}Y_n & \text{if } fn > 1 \end{array}$$

Symbols f and g, and constant c above correspond to constructor symbols associated to the normal forms of interest, which is more delicate in the presence of an order-sorted setting with kinds, as shown in the previous section.

When we have algebraic properties such as associativity, commutativity and identity, strict equality is not so well-studied and different formulations are possible; however we do not further discuss strict equality here because in the next section we are going to simplify it to just a rule `X =:= X` $\to$ `true`. Now we can search for equality with the desired behaviour, where `coin` is evaluated to `0` and, then, `=:=` checks that it is equal to `0` in order to reduce to `true`.

```
search coin =:= 0 =>* true .
Solution 1
empty substitution
No more solutions.
```

5.3 Conditional equations

Another relevant feature of functional logic programming languages is the use of conditional rules. The current implementation of narrowing in Maude deals only with unconditional rules but we transform conditional rules into unconditional ones using a standard technique in functional logic languages such as Curry (see [2, 26] for details). Conditional means that a rule has an extra element, apart from the left-hand and right-hand sides, which contains conditions and, intuitively, these conditions must be satisfied before the rule is applied. These conditions can be of different form in Maude but we restrict ourselves to just

[9] Strict not in the sense of argument evaluation but in the sense of checking only for values (normal forms).

[10] The symbol `=:=` is original from the system Curry where it is a built-in operation.

equality conditions. Note that modern functional logic programming languages use conditions of the form "t == t'" and "t =:= t'" for both syntactic and strict equality, respectively. However, we will simplify it here to conditions of the form "t =:= t'". In Maude syntax, a definition of the membership function for multisets of natural numbers is of the form:

```
sort NatSet .
op empty : -> NatSet . subsort Nat < NatSet .
op _;_ : NatSet NatSet -> NatSet [assoc comm id: empty] .
vars N E : Nat . var NS : NatSet .

op member : Nat NatSet -> [Bool] .
rl member(E, empty) => false .
crl member(E, N ; NL) => true if E =:= N == true .
```

Functional logic languages assume that, when there is a conditional rule, the only valuable case is usually when the condition is true and the case where the condition is false is irrelevant, due to problems of negation in functional logic programs [3]. We implement this idea in the paper. First, a new sort `Success` and a new constant symbol `success` are defined in Maude. Note that there is no symbol for the negative counterpart. Second, the `=:=` symbol is simply defined for the positive case returning `success`, e.g. for the sort `Nat` is defined as follows:

```
sort Success .
op success : -> Success [ctor] .
op _=:=_ : Nat Nat -> [Success] [comm] .
rl X:Nat =:= X:Nat => success .
```

Note that variable `X` must be of a specific sort but not a kind, in order to obtain the call-time choice semantics of Section 5.1. Third, we replace the conditional expression by a new operator ">>" with just the positive case, where we restrict reductions on the second argument using the attributes `frozen (2)` and `strat (1 0)`, as in the standard Maude `if-then-else-fi` symbol. Again, this transformation is standard in functional logic languages such as Curry, see [2, 26] where the symbol `>>` is also known as `if-then` (without an else branch).

```
op _>>_ : [Success] [Nat] -> [Nat] [frozen (2) strat (1 0)] .
rl success >> X:[Nat] => X:[Nat] .
```

Note that Maude expects conditions to be boolean expressions but we consider here conditions to be of sort `Success` just because we consider only the positive cases. The transformation of the previous definition of member is as follows:

```
op member : Nat NatSet -> [Bool] .
rl member(E, empty) => false .
rl member(E, N ; NS) => E =:= N >> true .
```

We can search for solutions to membership in the following form:

```
search member(N:Nat,0 ; s 0 ; s s 0) ~>! true .
Solution 1        Solution 2        Solution 3
N:Nat --> 0       N:Nat --> s 0     N:Nat --> s s 0
No more solutions.
```

5.4 Extra variables

As in logic programming, rules have extra variables not appearing in the left-hand side. There are different characterizations of rules with extra variables but we do not impose any restriction here, since extra variables are simply part of the right-hand side when using `>>` expressions. An interesting example is the definition of the function `last` returning the last element of a given list of natural numbers, but using the function append (`++`) instead of traversing the list to decompose the argument `NL` into a new list `NL'` and the last element `E`:

```
  op _++_ : NatList NatList -> [NatList] .
  rl nil ++ NL' => NL' .
  rl (N : NL) ++ NL' => N : (NL ++ NL') .

  op last : NatList -> [Nat] .
  rl last(NL) => NL' ++ (E : nil) =:= NL >> E [nonexec] .
```

Note that variables `E` and `NL'` are extra variables not appearing in the left-hand side and are quantified existentially. The `nonexec` label is necessary[11] because Maude accepts only rules and equations without extra variables. The execution of a query to `last` would be as follows, where we restrict to just the first solution:

```
search [1] in last(0 : s 0 : s s 0 : nil) ~>! X:Nat .
Solution 1
X:Nat --> s s 0
No more solutions.
```

5.5 Constraint solving and residuation

Logic programming is quite effective in solving goals and many strategies have been defined in the literature to speed up that process. A typical optimization in logic programming is to combine different goals where some of them are suspended until an instantiation is provided by another goal. This procedure is called *residuation* in functional logic programming languages such as Curry [26] and Escher [30] and it is easy to achieve in Maude. That is, residuation is based on the idea to delay or suspend function calls until they are ready for deterministic evaluation. Since the residuation principle evaluates function calls by deterministic reduction steps, nondeterministic search is usually encoded by predicates or disjunctions. Moreover, if some part of a computation might suspend, one

[11] A new attribute `extra-vars(E,NL')` may be used in the future to denote a rule with extra variables.

needs a primitive to execute computations concurrently. For instance, we include a symbol "`&`" for the conjunction of constraints using the sort `Success` so that both arguments are evaluated concurrently, i.e., if the evaluation of one argument suspends, the other one is evaluated. This requires very little in Maude.

```
  op _&_ : [Success] [Success] -> [Success] [assoc comm id: success] .
```

The trick in Maude is that we will be using equations, with a rewriting semantics, for those suspended function calls while we will be using rules, with a narrowing semantics in this paper, for the others (see Footnote 7).

For instance, if we define a predicate for generating natural numbers through narrowing:

```
  op nat : Nat -> [Success] .
  rl nat(0) => success .
  rl nat(s N) => nat(N) .
```

And change the former specification of addition to use equations, so that addition will be evaluated by rewriting (residuation) and never by narrowing:

```
  op _+_ : Nat Nat -> [Nat] .
  eq 0 + M = M .
  eq (s N) + M = s (N + M) .
```

Now we can solve the conjunction of two goals very effectively, since narrowing will be used only for `nat` and once this function instantiates some variable of an addition expression, the equations of addition will be used.

```
search [1] nat(X:Nat) & (X:Nat + 0) =:= s 0 ~>! success .
Solution 1
X:Nat --> s 0
No more solutions.
```

In the context of Maude, the narrowing search space is very much reduced, since only the evaluation of `nat` by narrowing generates new states and the evaluation of addition and equality is done by equations, which is very efficient in Maude. However, in many situations we have to reduce the search space by imposing an order of evaluation among different constraints, i.e., `Curry` provides a symbol `&>` that forces an evaluation of constraints from left to right, but this symbol is indeed our symbol `>>` defined above. Now we can run the previous query slightly faster.

```
search [1] nat(X:Nat) >> (X:Nat + 0) =:= s 0 ~>! success .
Solution 1
X:Nat --> s 0
No more solutions.
```

However, if we swap the constraints, `(X:Nat + 0) =:= s 0 >> nat(X:Nat)` does not return any value, whereas `(X:Nat + 0) =:= s 0 & nat(X:Nat)` does.

6 Executing the motivating example

The specification of the equational logic program in Section 2 by using the features and functionalities described in Section 5 is as follows.

```
mod MAC is
  pr SUCCESS . pr TRIPLIST . pr PSET .

  ops taylor helen william : -> Elem [ctor] .       ops umugu nzwawe amoc : -> Elem [ctor] .
  var L : TripList . var T : Trip . var PS : PSet .

  op gen : Elem -> [Success] .
  rl gen(taylor) => success .               eq gen(taylor) = success .
  rl gen(helen) => success .                eq gen(helen) = success .
  rl gen(william) => success .              eq gen(william) = success .
  rl gen(umugu) => success .                eq gen(umugu) = success .
  rl gen(nzwawe) => success .               eq gen(nzwawe) = success .
  rl gen(amoc) => success .                 eq gen(amoc) = success .

  op m0 : -> [PSet] .                       op c0 : -> [PSet] .
  eq m0 = taylor helen william  .           eq c0 = umugu nzwawe amoc  .

  op mset : PSet -> [PSet] .                op cset : PSet -> [PSet] .
  eq mset(PS) = PS /\ m0 .                  eq cset(PS) = PS /\ c0 .

  op boatok : Trip -> [Success] .           op boat : PSet -> Trip [ctor] .
  eq boatok(boat(X:Elem)) = gen(X:Elem) .
  eq boatok(boat(X1:Elem X2:Elem))
   = gen(X1:Elem) >> gen(X2:Elem) >> ((X1:Elem =/= X2:Elem) =:= true) .

  ops lb rb : TripList -> [PSet] .
  eq lb(nil) = m0 c0 .
  eq lb(L * boat(PS)) = if (even # L) then (lb(L) - PS) else (lb(L) PS) fi .
  eq rb(nil) = empty .
  eq rb(L * boat(PS)) = if (even # L) then (rb(L)  PS) else (rb(L) - PS) fi .

  op good : TripList -> [Success] .
  eq good(nil) = success .
  eq good(L * T)
   =  boatok(T) >> good(L) >> ( (# cset(lb(L * T)) =< # mset(lb(L * T))
                                  or (# mset(lb(L * T)) == 0))
                                and
                                (# cset(rb(L * T)) =< # mset(rb(L * T))
                                  or (# mset(rb(L * T)) == 0))            ) =:= true .

  op solve : TripList -> [Success] .
  eq solve(L) = good(L) >> (lb(L) == empty) =:= true .
endm
```

We have used a specific generator function `gen` which is the only function using rules apart from `=:=`; and both have rules and equations, both for narrowing and rewriting (residuation). Predicates are indeed considered as conditional equations evaluated into `success` and the conditions have been transformed using the `>>` operator. The remaining code is encoded into equations, thus speeding up the execution. We have also used `>>` to order the evaluation of constraints, thus speeding up the execution too. We omit[12] the specification of the auxiliary modules for list of trips and sets of missionaries and cannibals, but they are defined with equations and following the description of this paper.

One of the possible solutions is

[12] The experiments are available at `http://www.dsic.upv.es/~sescobar/MAC.html`.

```
search  solve(nil * boat(taylor umugu) * boat(taylor) * boat(nzwawe amoc) * boat(umugu) *
                    boat(william helen) * boat(helen nzwawe) * boat(taylor helen) *
                    boat(amoc) * boat(umugu amoc) * boat(helen) * boat(helen nzwawe) )
            =>! success .
Solution 1
empty substitution
No more solutions.
```

We can search for solutions to queries using variables, for instance the person chosen in the last two steps, e.g. `helen`, is irrelevant and we use variable `E`:

```
search solve(nil * boat(taylor umugu) * boat(taylor) * boat(nzwawe amoc) * boat(umugu) *
                  boat(william helen) * boat(helen nzwawe) * boat(taylor helen) *
                  boat(amoc) * boat(umugu amoc) * boat(E) * boat(E nzwawe) ) ~>! success .
Solution 1       Solution 2       Solution 3        Solution 4       Solution 5
E:Elem --> amoc  E:Elem --> helen  E:Elem --> taylor  E:Elem --> umugu  E:Elem --> william
No more solutions.
```

The cannibal chosen in steps 9th and 10th is irrelevant and we use variable `E1`:

```
search  solve(nil * boat(taylor umugu) * boat(taylor) * boat(nzwawe amoc) * boat(umugu) *
                    boat(william helen) * boat(helen nzwawe) * boat(taylor helen) *
                    boat(E1) * boat(umuguE1) * boat(E) * boat(E nzwawe) ) ~>! success .
Solution 1                                 Solution 2
E1:Elem --> amoc ; E:Elem --> amoc       E1:Elem --> amoc ; E:Elem --> helen
Solution 3                                 Solution 4
E1:Elem --> amoc ; E:Elem --> taylor     E1:Elem --> amoc ; E:Elem --> umugu
Solution 5                                 Solution 6
E1:Elem --> amoc ; E:Elem --> william    E1:Elem --> nzwawe ; E:Elem --> amoc
Solution 7
E1:Elem --> nzwawe ; E:Elem --> umugu
No more solutions.
```

The current implementation of narrowing as symbolic reachability in Full Maude is not able to handle the most general and powerful query `search solve(L) ~>* success` due to its huge execution time and big memory consumption.

The well-versed reader of narrowing features in Maude may wonder whether this equational logic program can be used with the latest variant-generation features recently available in Maude [12]. The answer is yes. First, this program has no nondeterministic computation and no extra variables in right-hand sides, so it can be turned into a confluent, terminating and coherent equational theory modulo associativity, commutativity, and an identity symbol. Then, variant generation can be tried on the different input terms of the search commands shown above and the generated variants contain the expected results shown above. Indeed, this program and the output can be found in the url given above. This process runs faster than symbolic reachability, but the results are the same and are not discussed in this paper. Note that the expression `solve(L)` does not have a finite number of most general variants and, thus, symbolic reachability would be able to find the solution given enough time and memory while the variant generation is incapable of finding it.

7 Conclusions

We have tried to illustrate how concurrent constraint functional logic programs can be written and executed by using the novel infrastructure of narrowing as symbolic reachability in Maude. This paper shows how Maude goes beyond standard practices in the functional logic area by using, e.g. equational properties such as associativity and commutativity or an order-sorted setting. A detailed comparison of features that are available in `Curry` or `Maude` is outside of this paper, as well as a performance comparison between both languages.

References

1. M. Alpuente, D. Ballis, and M. Falaschi. Transformation and debugging of functional logic programs. In A. Dovier and E. Pontelli, editors, *25 Years GULP*, volume 6125 of *Lecture Notes in Computer Science*, pages 271–299. Springer, 2010.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40:875–903, 2005.
3. S. Antoy. Programming with narrowing: A tutorial. *Journal of Symbolic Compututation*, 45(5):501–522, 2010.
4. S. Antoy, R. Echahed, and M. Hanus. Parallel evaluation strategies for functional logic languages. In L. Naish, editor, *ICLP*, pages 138–152. MIT Press, 1997.
5. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.
6. S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.
7. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In M. Proietti, editor, *Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, volume 1048 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 1996.
8. K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
9. M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in maude 2.4. In R. Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.
10. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science.* Springer, 2007.
11. N. Dershowitz. Goal Solving as Operational Semantics. In *International Logic Programming Symposium (Portland, OR)*, pages 3–17, Cambridge, MA, December 1995. MIT Press.
12. F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. L. Talcott. Variants, unification, narrowing, and symbolic reachability in maude 2.6. In M. Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.
13. F. Durán and J. Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *J. Log. Algebr. Program.*, 81(7-8):816–850, 2012.
14. R. Echahed and N. Peltier. Narrowing data-structures with pointers. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do*

Norte, Brazil, September 17-23, 2006, Proceedings, volume 4178 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2006.

15. S. Escobar. Refining weakly outermost-needed rewriting and narrowing. In *PPDP*, pages 113–123. ACM, 2003.
16. S. Escobar. Implementing natural rewriting and narrowing efficiently. In Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2004.
17. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In F. Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.
18. S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2005.
19. S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.
20. M. Fay. First-order unification in an equational theory. In W. H. Joyner, editor, *Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas, USA*, pages 161–167. Academic Press, 1979.
21. J. Goguen and J. Meseguer. Eqlog: Equality, Types and Generic Modules for Logic Programming. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.
22. J. A. Goguen and J. Meseguer. Equality, types, modules, and (why not ?) generics for logic programming. *J. Log. Program.*, 1(2):179–210, 1984.
23. J. A. Goguen and J. Meseguer. Models and equality for logical programming. In H. Ehrig, R. A. Kowalski, G. Levi, and U. Montanari, editors, *TAPSOFT*, volume 250 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 1987.
24. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
25. M. Hanus. Multi-paradigm declarative languages. In V. Dahl and I. Niemelä, editors, *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 45–75. Springer, 2007.
26. M. Hanus. Functional logic programming: From theory to curry. In A. Voronkov and C. Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 123–168. Springer, 2013.
27. S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Computer Science.* Springer, 1989.
28. J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. A. Kowalski, editors, *Fifth Conference on Automated Deduction, CADE 1980, Les Arcs, France, July 8-11, 1980, Proceedings*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.
29. J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15(4):1155–1194, 1986.
30. J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), 1999.
31. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
32. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

33. J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Proceedings of the Third International Conference (ALP'92)*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200, Berlin, 1992. Springer-Verlag.
34. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.
35. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
36. J. C. G. Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.
37. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
38. D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51:277–289, 2001.
39. U. S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proceedings of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE Computer Society Press, 1985.
40. A. Riesco and J. Rodríguez-Hortalá. Singular and plural functions for functional logic programming. *CoRR*, abs/1203.2431, 2012.
41. J. R. Slagle. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *J. ACM*, 21(4):622–642, 1974.
42. TeReSe, editor. *Term Rewriting Systems.* Cambridge University Press, Cambridge, 2003.
43. L. Vigneron. Automated deduction techniques for studying rough algebras. *Fundamenta Informaticae*, 33(1):85–103, 1998.