Extensions of Logic Programming in Maude*

Santiago Escobar

VRAIN, Universitat Politècnica de València
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
sescobar@upv.es

Abstract. In 2022, the logic programming community celebrated the milestone of 50 years of evolution of logic programming languages, started in 1972 with the first version of Prolog. In this paper, we consider how extensions of logic programming can be handled in the High-Performance Logical Framework Maude.

1 Introduction

Last year, the logic programming community celebrated the milestone of 50 years of evolution of logic programming languages, started in 1972 with the first version of Prolog [54]. Logic programming distinguishes from other programming paradigms such as traditional imperative programming by adding free (logical) variables and search, paving the way for inference engines that are measured in terms of inferences per second rather than calculations per second. Moreover, logic programming distinguishes from other declarative paradigms such as functional programming by using, again, free variables and search but also by using term unification instead of term matching.

The paradigm of logic and functional programming (see [41,42,43,49] and references therein) combines both functional programming and logic programming in different ways. From the functional programming perspective, it borrows algebraic data types, advanced typing, evaluation strategies, and higher-order functions among other features; allowing programmers to have nested expressions and, thus, normalization strategies. From the logic programming perspective, it borrows logical variables, computing with partial information, constraint solving, and nondeterministic search for solutions among other features; allowing programmers to have search mechanisms with local and global control. Many researchers in the functional logic programming area (see [71,38,19,62,45]) have tried, since the eighties, to combine the best features of both paradigms into a single language, Curry [44] being a distinguished instance, and many possibilities have been explored (see [45,47] for a survey). Nowadays there is a remarkable

*This work has been partially supported by the EC H2020-EU grant agreement No. 952215 (TAILOR), by the grant PID2021-122830OB-C42 funded by MCIN/AEI/10.13039/501100011033 and ERDF A way of making Europe, and by the grant PCI2020-120708-2 funded by MICIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR.

body of programming languages and tools in the functional logic area as a result of these efforts.

In [25,31], we showed how functional-logic programs written in Curry can be transformed into the high-performance modelling and programming language Maude [20,16] by relying on the advanced equational unification capabilities, for any combination of associativity, commutativity and identity, and the distinction between equations and rules in Maude. In [20], we specified a simple logic programming language in Maude illustrating various (logical) computation features. In this paper, we consider how some extensions of logic programming, as the ones discussed in [54,18], can be handled in Maude.

Modern multi-paradigm programming languages [48] have embarked on combining different paradigms in a seamless way: functional programming, logic programming, concurrent programming, and constraint programming. Curry [44] offers most of these features but other logic and/or functional programming systems too, such as the Ciao Prolog [50] or even Maude [21,23]. Indeed, the Eqlog programming language [36] developed by Joseph Goguen and José 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. It was envisaged to embed order-sorted equational logic and Horn logic without equality into a suitable Horn logic with equality [37]. Indeed, in [23, Sec 8.1], a simple but fully executable interpreter of Eqlog was provided, although the program transformations of [25,31] are more effective in general. Since the eighties, José Meseguer has been interested in including logical features into Maude (see his paper [63] dedicated to Goguen's 65th birthday) but most of the appropriate technology was missing and has been developed recently (see [65,67]). Many people strongly believe that Maude *with logical features* would be an excellent choice in the near future for multi-paradigm programming: equational programming, object-oriented programming, concurrent programming, logic programming, execution strategies, symbolic computation, and constraint solving; combined with reflection and a suite of many different formal tools such as model checkers and theorem provers.

Many alternatives for multi-paradigm programming have attempted to extend a logic language with functional syntax (Boolean equations rather than just predicates, nested expressions instead of functors) and unfold expressions into flattened predicates with extra variables. The use of coroutines allows a finer control over evaluated predicates in logic programming but may yield incompleteness problems and an infinite search space in many situations [46]. 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 [74], then used as a mechanism for solving equational unification problems [35]. It became the de facto evaluation mechanism for functional logic programming languages [7], and it was generalized from equational unification problems to solve the more general problem of symbolic reachability [68] and, in a more modern perspective, of *logical* model checking in [32,12]. The narrowing mechanism has a number of important applications including automated proofs of termination [11], execution of functional logic programming languages [7], program transformation [3], program debugging [1,4], partial evaluation [5,6,2], verification of cryptographic protocols [68], equational unification [51], and narrowing-based SMT solving [75,66], just to mention a few.

The work on the Maude-NPA protocol crypto analyzer [27,26,39,9,8,10] is the most impressive use of narrowing-based symbolic reachability analysis in Maude to date and has served as inspiration to many other researchers, tools and techniques, e.g. other crypto protocol analyzers such as Tamarin [61] and AKISS [14].

In Section 2, we will recall some of the feature of the High-performance Logical Framework Maude. In Section 3, we discuss some extensions of logic programming and provide how relevant program verification examples published in [18] can be easily modelled and verified in Maude. We conclude in Section 4 with some thoughts on future work.

2 Maude

Maude [20,16] is a high-level programming language and system that supports functional, concurrent, logic, and object-oriented computations. A Maude rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ combines a set R of term rewrite rules, which specify the concurrent transitions of a system, with an equational theory $\mathcal{E}$ that specifies the algebraic datatypes of the system's states. The equational theory $\mathcal{E}$ is split into a set E of equations and a set B of axioms. The axioms B are equalities representing algebraic laws such as associativity (A), commutativity (C), and unit symbols (U). The equations E are implicitly oriented from left to right as rewrite rules and operationally used as simplification rules modulo the axioms B (see [64] for further details). The rewrite rules R are applied to terms by matching[1] modulo the equations E and the axioms B. When narrowing instead of rewriting is performed, rewrite rules are applied by unification modulo the equations E and the axioms B. Recently, it has been endowed with logical features, such as equational unification and symbolic reachability [21,65,67,23].

2.1 Equational Unification

The most recent Maude 3.2.1 release [16] provides efficient, terminating and complete unification procedures. The most basic is order-sorted B-unification, where B is any combination of associativity and/or commutativity and/or unit element axioms. Note that if a symbol is associative but not commutative, Maude's algorithms are optimized to favor many commonly occurring cases where typed A-unification is finitary, and provide a finite set of solutions and an incompleteness

[1] This is conceptually exact; but operationally, exploiting a property called *coherence* [22], rules R can be applied modulo the axioms B only.

warning outside such cases (see [24]). The most general version is order-sorted $E \cup B$-unification in the *user-definable* infinite class of theories $E \cup B$ satisfying the *finite variant property* (FVP) [17,28].

Let us consider the following equational theory from [23] for the Booleans (with self-explanatory, user-definable syntax):

```
fmod BOOL-FVP is protecting TRUTH-VALUE .
   op _and_ : Bool Bool -> Bool [assoc comm] .
   op _xor_ : Bool Bool -> Bool [assoc comm] .
   op not_ : Bool -> Bool .
   op _or_ : Bool Bool -> Bool .
   op _<=>_ : Bool Bool -> Bool .
   vars X Y Z W : Bool .

   eq X and true = X [variant] .
   eq X and false = false [variant] .
   eq X and X = X  [variant] .
   eq X and X and Y = X and Y [variant] .   *** AC extension
   eq X xor false = X [variant] .
   eq X xor X = false [variant] .
   eq X xor X xor Y = Y [variant] .         *** AC extension
   eq not X = X xor true [variant] .
   eq X or Y = (X and Y) xor X xor Y [variant] .
   eq X <=> Y = true xor X xor Y [variant] .
endfm
```

The axioms B are the associativity-commutativity (AC) axioms for `xor` and `and` (specified with the `assoc comm` attributes). The equations E are terminating and confluent modulo B. Two equations are added to achieve strict B-*coherence* [64]. The remaining equations in E define `or`, `not` and `<=>` as definitional extensions. The `variant` attribute declares that the equation will be used for *folding variant narrowing* [28]. This is a narrowing strategy applying oriented equations modulo axioms that is terminating and complete for equational theories that are FVP and, even more, optimally terminating in the sense that no other narrowing strategy could compute fewer variants and still be complete. Indeed, the theory specified by the functional module `BOOL-FVP` is FVP; see [16] for further details on how to check this property.

A complete, finite set of $E \cup B$-unifiers can be computed with Maude's `(filtered) variant unify` command. For our `BOOL-FVP` example, it gives us a Boolean satisfiability decision procedure. Such a procedure cannot compete with mainstream SAT-solvers but illustrates with a simple example how unification commands provide an off-the-shelf SAT solver (see [23]).

```
Maude> filtered variant unify (X or Y) <=> Z =? true .
rewrites: 3224 in 12765ms cpu (14776ms real) (252 rewrites/second)

Unifier 1
X --> #1:Bool xor #2:Bool
Y --> #1:Bool
```

```
Z --> #2:Bool xor (#1:Bool and (#1:Bool xor #2:Bool))

No more unifiers.
Advisory: Filtering was complete.
```

Fresh, newly generated variables follow the form `#1:Bool`.

Let us consider another example from [66] defining Presburger arithmetic of the natural numbers with addition and comparison that imports `BOOL-FVP`:

```
fmod NAT-FVP is protecting BOOL-FVP .
    sorts Nat NzNat .
    subsort NzNat < Nat . ---Non-zero naturals

    op 0 : -> Nat [ctor] .
    op 1 : -> NzNat [ctor] .
    op _+_ : NzNat Nat -> NzNat [ctor assoc comm id: 0] .
    op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
    vars X Y : Nat . var Z : NzNat .

    op _>_ : Nat Nat -> Bool .
    eq X + Z > X = true [variant] .
    eq X > X + Y = false [variant] .
endfm
```

The axioms B are the associativity, commutativity, and identity (ACU) axioms for the addition, where 0 is the identity element. For example, the natural number 3 is written $1+1+1$. The equations E defining comparison are terminating and confluent modulo B. The theory specified by the functional module `NAT-FVP` is also FVP. In this case, it gives us a Presburger arithmetic satisfiability decision procedure. Again, such a procedure cannot compete with mainstream SMT-solvers for Presburger arithmetic but illustrates with a simple example how unification commands provide an off-the-shelf SMT solver (see [23]). Indeed, fairly complex formulas can be proved, e.g. $\forall X, Y : X + Y > Y \Leftrightarrow Y > 0$ can be represented as an unification problem $(\exists X, Y : X + Y > Y \Leftrightarrow Y > 0) =^{?}$ `false`:

```
Maude> filtered variant unify (X + Y) > X <=> Y > 0 =? false .
rewrites: 10 in 1ms cpu (1ms real) (8726 rewrites/second)

No unifiers.
Advisory: Filtering was complete.
```

2.2 Symbolic Reachability

When the rewrite theory $\mathcal{R}$ is *topmost*, meaning that the rules R rewrite the entire state, narrowing with rules R modulo the equations $\mathcal{E}$ is a *complete* symbolic reachability analysis method for *infinite-state systems* [68]. That is, given a term u with variables $\overrightarrow{x}$, representing a typically infinite set of initial states, and another term v with variables $\overrightarrow{y}$ (probably sharing some variables with $\overrightarrow{x}$), representing a possibly infinite set of target states, narrowing can answer the question: *can an instance of u reach an instance of v?* That is, does the formula $\exists \overrightarrow{x}, \overrightarrow{y} \;\; u \rightarrow^* v$ hold in $\mathcal{R}$? Note that, if the *complement* of a system invariant I can be symbolically described as the set of ground instances of terms in a set $\{v_1, \ldots, v_n\}$ of pattern terms, then narrowing with rules R modulo the equations $\mathcal{E}$ provides a semi-decision procedure for verifying whether the system specified by $\mathcal{R}$ fails to satisfy I starting from an initial set of states specified by u. Namely, I holds iff no instance of any v_i can be reached from some instance of u. Moreover, if the narrowing-based reachability graph is finite, then narrowing provides a decision procedure for verifying invariants, as shown in the examples below.

The `vu-narrow` command implements narrowing with $\mathcal{R}$ modulo $E \cup B$ by performing $E \cup B$-unification at each narrowing step. However, the number of symbolic states that need to be explored can be *infinite*. This means that if no solution exists for the narrowing search, Maude will search forever, so that only *depth-bounded searches* will terminate. However, Maude implements some form of tabling with the `{fold} vu-narrow {filter,delay}` command that performs a powerful *symbolic state space reduction* by: (i) removing a newly explored symbolic state v' if it $E \cup B$-matches a previously explored state v and replacing a transition with target v' by transitions with target v; and (ii) using minimal sets of $E \cup B$-unifiers for each narrowing step and for checking common instances between a newly explored state and the target term (ensured by words `filter` and `delay`). This can make the entire search space finite and allows full verification of invariants for some infinite-state systems.

Consider the following Maude specification of Lamport's bakery protocol that extends the specification of [23] with extra variables in right-hand sides and conditional[2] rules. The `narrowing` attribute declares that the rule will be used only for folding narrowing with rules modulo the whole equational theory.

```
mod BAKERY-EXTRAVAR is
  pr NAT-FVP .
  sorts LNat Nat? State WProcs Procs .
  subsorts Nat LNat < Nat? .  subsort WProcs < Procs .
  op [_] : Nat -> LNat .               *** number-locking operator
  op < wait,_> : Nat -> WProcs .
  op < crit,_> : Nat -> Procs .
  op mt : -> WProcs .                  *** empty multiset
  op __ : Procs Procs -> Procs [assoc comm id: mt] .    *** union
  op __ : WProcs WProcs -> WProcs [assoc comm id: mt] . *** union
  op _|_|_ : Nat Nat? Procs -> State .

  vars n m i j k : Nat . var x? : Nat? . var PS : Procs . var WPS : WProcs .
  var z : NzNat .
  crl [new]: m | n | PS => m + z | n | < wait,m > PS
     if m > n [narrowing] .
```

[2] Maude does not currently accept conditional equations or conditional rules for any form of narrowing. However, the transformation of conditional rules into unconditional rules of [56] can be applied to this example.

```
  crl [enter]: m | i | < wait,j> PS => m | [j] | < crit,j> PS
     if m > i and m > j and not (i > j) [narrowing] .
  crl [leave]: m | [n] | < crit,n > PS => m | n + z | PS
     if m > n + z [narrowing] .
  crl [lost]: m | i | < wait,j > PS => m + z | i | < wait,m > PS
     if (m > i) and (i > j) [narrowing] .
endm
```

The states of `BAKERY-EXTRAVAR` have the form "`m | x? | PS`" with `m` the ticket-dispensing counter, `x?` the (possibly locked) counter to access the critical section, and `PS` a multiset of processes either waiting or in the critical section. This rewrite theory is an infinite state system in different ways: the rule labelled `[new]` creates new processes, and the counters `m` and `x?` can grow unboundedly. When a waiting process enters the critical section by applying the rule labelled `[enter]`, the second counter `n` is locked and written `[n]`. The locked process is removed and the second counter is unlocked and incremented when the rule labelled `[leave]` is applied.

This example differs from the one in [23] in the use of extra variables. First, the general invariant that the ticket-dispensing counter must always be greater than the counter to access the critical section is checked and preserved by all the rules. Second, when creating new processes, the ticket-dispensing counter can be increased in any amount instead of just one unit, thanks to the non-zero variable `z` appearing only in the righthand side of the rule. Third, because of such unbounded increment of the ticket-dispensing counter, the counter to access the critical section is not sequential and, thus, the rule labelled `[enter]` needs to check that the counter of a waiting process is greater or equal to the counter to access the critical section; we also check the general invariant that it is smaller than the current ticket-dispensing counter. Fourth, when removing a completed critical process using the rule labelled `[leave]`, the counter to access the critical section can be increased in any amount instead of just one unit, thanks to the non-zero variable `z` appearing only in the righthand side of the rule, while ensuring the general invariant. Fifth, it is possible that a process missed the critical section because of the unbounded increment of the counter to access the critical section and we have added a new rule labelled `[lost]` that takes a new ticket.

The key invariant is *mutual exclusion*. Note that the term "`i | x? | < crit, j > < crit, k > PS`" describes all states in the *complement* of mutual exclusion states. Without the `fold` option, narrowing does not terminate, but with the following command we can verify that `BAKERY-EXTRAVAR` satisfies mutual exclusion for the much more general *infinite* set of initial states with waiting processes "`m | n | WPS`". Note that this cannot be achieved by standard rewriting-based reachability analysis from an initial state such as "`0 | 0 | mt`" because of the use of extra variables in the righthand sides of the rules.

```
Maude> {fold} vu-narrow {delay, filter}
     m | n | WPS =>* i | x? | < crit,j > < crit,k > PS .
```

```
No solution.
rewrites: 145 in 78430ms cpu (86933ms real) (1 rewrites/second)
```

The *finite* folded narrowing space is displayed in Figure 1. The verified property is very strong, since mutual exclusion is proved for an unbounded number of processes.

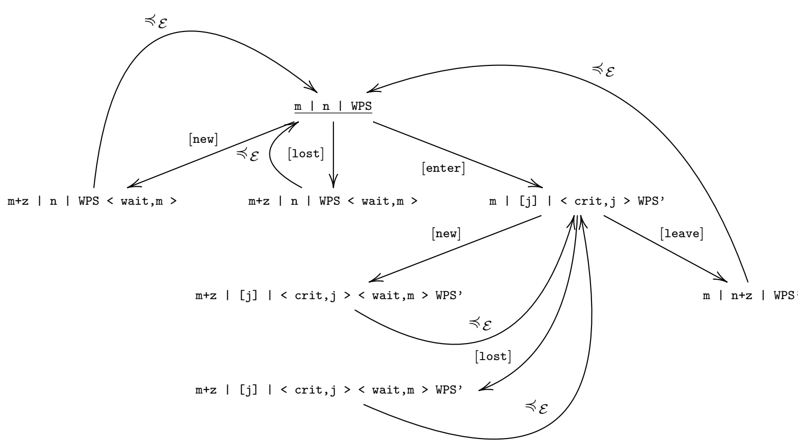

Fig. 1: Folded Narrowing Space for `BAKERY-EXTRAVAR`

3 Extensions of Logic Programming in Maude

Logic programming has inspired many related areas and, as a result, several different languages have created their own communities on extensions of logic programming. Let us recall some of these extensions.

Strategies Both logic and functional paradigms consider evaluation strategies. In the case of functional programming better performance can be achieved by either using eager evaluation such as OCaml [69] and Maude [20] or lazy evaluation such as Haskell [53]. In the case of Prolog, it provides a sequential, depth-first, deterministic exploration of the proof tree generated by SLD resolution, thanks to backtracking, selecting clauses in a top-down manner, and selecting predicates in a left-to-right order.
Maude allows operator strategy annotations for rewriting evaluation of oriented equations [58,59,40] and a versatile strategy rewriting language for rules [73,16]. In the case of narrowing, the folding variant narrowing [28] is a very specialized strategy for oriented equations modulo axioms that it is neither eager nor lazy[3]. It prioritizes simplification rewriting steps over narrowing steps; indeed it prioritizes narrowing steps with more general computed substitutions. This narrowing

[3] Forms of lazy evaluation have been developed in [34,33,30,29] for Maude.

strategy discards narrowing steps where the computed substitution is not normalized. For narrowing with rules, there is no strategy narrowing language yet as the one already developed for rewriting with rules, although folding, as shown above, reduces dramatically the search space.

Tabling Tabling is a refinement of the SLD resolution incorporated to many Prolog systems that consists on maintaining a table of subgoals that are invoked during execution, along with their answers if they have been computed. If the current subgoal is present in the table, its evaluation is not attempted and their answers are reused. Tabled logic programs terminate in many more situations than standard logic programs, although, from a theoretical perspective, both compute the same answers.
We have shown in the previous section how tabling has been incorporated into narrowing with equations. It is an essential feature for an equational theory being FVP, and narrowing with rules, via folding narrowing with rules.

Coroutining Coroutining ensures that a predicate is selected only if it is fully instantiated. Coroutining is a key element in current logic programming systems by improving the control the user has over the search tree. Logic programs should be independent of the selection criteria in the SLD resolution but, from a practical perspective, some evaluation order over the predicates may be desirable in terms of efficiency, termination, or the intended answers.
In Maude, we have several features that can be intelligently exploited. Equations applied for rewriting are not labelled with the `variant` attribute and are used just for simplification before any other action is taken. Equations applied for narrowing are labelled with the `variant` attribute and combined for simplification with those without the `variant` attribute. Rules applied only for narrowing are labelled with the `narrowing` attribute and are the only ones used by the `vu-narrow` command. Rules without the `narrowing` attribute are only used by rewriting-based commands such as `rewrite`, `frewrite`, and `search`.

Constraints When considering problems beyond syntactic unification of two terms, constraints given in richer domains provide a very flexible programming and solving language framework. Constraint Logic Programming (CLP) was presented in the landmark paper [52] parameterized by the constraint domain. The key insight was to generalize syntactic term unification into constraint solving over a specific semantic domain. In this way, traditional logic programming can be understood as $CLP(\mathcal{H})$ where $\mathcal{H}$ denotes the equalities over Herbrand terms. The CLP framework was first instantiated as $CLP(\mathcal{R})$ for linear equations and inequations over real numbers using Gaussian elimination. For software verification, Constrained Horn Clauses (CHCs) is more common than CLP, in the sense that software verification problems can be achieved using CHC logic programming techniques.
Maude has been extensively used for software verification, see [60,15]. Furthermore, rewriting with mainstream SMT solvers has been used for software verification, see [72,13,70,55]. However, narrowing in Maude has not been a popular

topic of interest for software verification. The two examples below motivate further uses of narrowing for software verification by making use, as in Section 2.2, of a property verification via not finding the complement of the property in a finite-state narrowing-based state space generated using transition rules and constraint solving via variant-based equational unification. However, we have also developed in [57] a framework for narrowing-based symbolic reachability combined with mainstream SMT solvers, which provides an alternative, complementary approach to the one presented here and which will be further developed in the future. Note that rewriting-based reachability combined with mainstream SMT solvers is already available in Maude, see [16].

3.1 Software Verification using Program Semantics

In [18], it is described how software verification is the encoding of a verification problem in CHC form. For instance, the imperative program of Figure 2 is translated into a logic program and the Hoare triple $\{m \geq 0\} sum = sum\_upto(m) \{sum \geq m\}$ is satisfied only if the corresponding logic program is satisfiable; we omit such a logic program but it is available in [18].

```
int sum_upto(int x) {
  int r = 0 ;
  while (x > 0) {
    r = r + x;  x = x - 1; }
  return r;
}
```

Fig. 2: Imperative program fragment

A simple imperative interpreter for this syntax can be defined in Maude as follows using a continuation-style very similar to the K semantics [15]. Each configuration of the interpreter has the form "`Program | Memory`" where the memory is a set of bindings from variable names to natural numbers. Expression evaluation consists in pushing and popping partially evaluated elements into the program using the semicolon as a stacking operator. We omit sort information and some operator definitions and show just the transitions associated to the operational semantics. We import the previous `NAT-FVP` module but rename its addition and comparison operators to avoid conflicts with the additions and comparison operations of the new syntax.

```
mod CHC is protecting NAT-FVP * (op _+_ to _++_, op _>_ to _>>_) .
  ...
  eq (nat V) ; P | M = P | (M (V -> 0))   .              --- New Variable
  eq (V = E) ; P | M = E ; (V = {}) ; P | M  .           --- Assignment
  eq N ; (V = {}) ; P | M  = P | (M  (V -> N)) .         --- Cont'd
  eq V ; P | (M  (V -> N)) = N ; P | (M  (V -> N)) .     --- Variable
  eq (E1 > E2) ; P | M = E1 ; E2 ; > ; P | M .           --- Comparison
```

```
  eq N ; E2 ; > ; P | M = E2 ; N ; > ; P | M .          --- Cont'd
  eq N2 ; N1 ; > ; P | M = (N1 >> N2) ; P | M .         --- Cont'd
  eq (E1 + E2) ; P | M  = E1 ; E2 ; + ; P | M .         --- Addition
  eq N ; E2 ; + ; P | M = E2 ; N ; + ; P | M .          --- Cont'd
  eq N2 ; N1 ; + ; P | M = (N1 ++ N2) ; P | M .         --- Cont'd
  eq E - 1 ; P | M  = E ; - ; P | M  .                  --- Predecessor
  eq N ; - ; P | M   = pred(N) ; P | M .                --- Cont'd
  eq while E {B} ; P | M = E ; while E {B} ; P | M .    --- While
  rl true ; while E {B} ; P | M => B ; while E {B} ; P | M [narrowing] .
  rl false ; while E {B} ; P | M => P | M [narrowing] .
endm
```

Note that we have defined only as narrowing rules the two alternatives associated to the conditional expression of a loop. All the other transitions are defined as equations without the `variant` attribute in order to reduce the folded narrowing search space shown in Figure 3 below. This is safe under the assumption that only logical variables over the natural domain will appear in the terms of a `vu-narrow` command. This is related to rewriting with SMT in Maude, where the boolean expression of the while loops would be expressed as a term sent to an SMT solver for satisfiability but it is not comparable to the verification performed below, since constraint solving rather than satisfiability is actually being performed and a finite narrowing-based search graph is obtained.

The intended invariant here, as described above, is that the result of the program is greater than the original argument. The imperative program is simplified into the following initial configuration of the program semantics `while (x > 0) {r = r + x ; x = x - 1} | (x -> X ++ Z) (r -> R)` where `X` and `R` are *logical* variables of sort `Nat` but `Z` is a *logical* variable of sort `NzNat`, since the Hoare triple assumed the argument was greater than or equal to 0 and we are going to search for the complement of the invariant. Note that the target pattern "`skip | (x -> W) (r -> X)`" contains a new logical variable `W` but reuses the previous logical variable `X` in order to describe all the states in the *complement* of the invariant, i.e. the original argument is `X ++ Z` but the result is `X`.

```
Maude> {fold} vu-narrow {delay, filter}
  while (x > 0) {r = r + x ; x = x - 1} | (x -> X ++ Z) (r -> R)
  =>*
  skip | (x -> W) (r -> X) .

No solution.
rewrites: 79 in 16ms cpu (19ms real) (4725 rewrites/second)
```

The *finite* folded narrowing space is displayed in Figure 3. Note that the root node is the original source term but normalized with the equations. In contrast to the CHC approach relying on logic programming, we are able to verify this property without any artificial encoding, just in a very natural way.

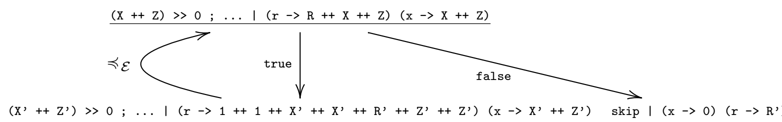

Fig. 3: Folded Narrowing Space for `CHC`

3.2 Software Verification on Algebraic Data Types

In [18], it is also described how software verification beyond decidable fragments accepted by mainstream SMT solvers can be translated into the encoding of a verification problem in CHC form. For instance, the Tree-Processing program, written in OCaml syntax, in Figure 4 is translated into a logic program and the property

$$\forall n, t : n \geq 0 \Rightarrow \mathit{min\text{-}leafdepth}(\mathit{left\text{-}drop}(n,t)) + n) \geq \mathit{min\text{-}leafdepth}(t) \qquad (1)$$

is satisfied only if the corresponding logic program is satisfiable; we omit such a logic program but it is available in [18].

```
type tree = Leaf | Node of int * tree * tree ;;
let min x y = if x < y then x else y ;;
let rec min-leafdepth t = match t with
     | Leaf -> 0
     | Node(x,l,r) -> 1+min(min-leafdepth(l),min-leafdepth(r)) ;;
let rec left-drop n t = match t with
    | Leaf -> Leaf
    | Node(x,l,r) -> if n <= 0 then Node(x,l,r) else left-drop (n-1) l ;;
```

Fig. 4: OCaml program fragment

A simple functional interpreter for this syntax can be defined in Maude as follows. There is no need for a general configuration of the interpreter and the two functional operations are directly translated into Maude operators. More sophisticated approaches are clearly possible but we choose the simplest one to ease the presentation.

```
mod TREE is protecting NAT-FVP .
  sort Tree .
  op Leaf : -> Tree .
  op Node : Nat Tree Tree -> Tree .
  vars N M : Nat . vars T L R : Tree .

  op minLD : Tree -> Nat .
  eq minLD(Leaf) = 0 .
  eq minLD(Node(N,L,R)) = 1 + min(minLD(L),minLD(R)) .
```

```
  rl minLD(Leaf) => 0 [narrowing] .
  rl minLD(Node(N,L,R)) => 1 + min(minLD(L),minLD(R)) [narrowing] .

  op leftDrop : Nat Tree -> Tree .
  eq leftDrop(N,Leaf) = Leaf .
  eq leftDrop(0,Node(M,L,R)) = Node(M,L,R) .
  eq leftDrop(N + 1,Node(M,L,R)) = leftDrop(N,L) .
  rl leftDrop(N,Leaf) => Leaf [narrowing] .
  rl leftDrop(0,Node(M,L,R)) => Node(M,L,R) [narrowing] .
  rl leftDrop(N + 1,Node(M,L,R)) => leftDrop(N,L) [narrowing] .
endm
```

Note that we have duplicated the narrowing rules as equations in order to collapse terms that are semantically equivalent but also to reduce the folded narrowing search space shown in Figure 5 below.

In this case, we do not have an invariant and, thus, we are not targeting the complement of an invariant but we transform property (1) into an existential expression targeting `false`, in order to prove that the universal theorem holds.

```
Maude> {fold} vu-narrow {delay, filter}
  not (minLeafDepth(T) > (minLD(leftDrop(N,T)) + N))  =>* false .

No solution.
rewrites: 19 in 1ms cpu (1ms real) (12541 rewrites/second)
```

The *finite* folded narrowing space is displayed in Figure 5. Note that the root node is the original source term but normalized with the equations. Again, in contrast to the CHC approach relying on logic programming, we are able to verify this property without any artificial encoding, just in a very natural way.

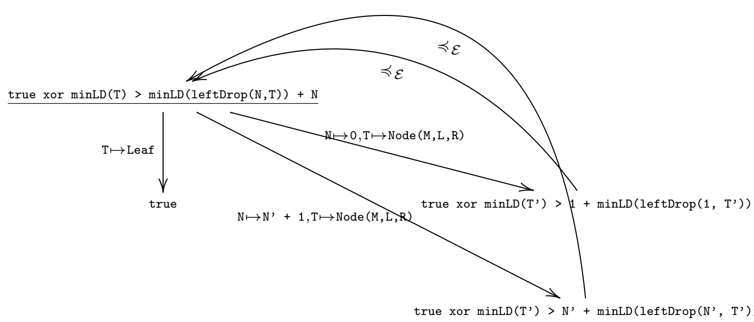

Fig. 5: Folded Narrowing Space for `OCaml`

4 Conclusions

As explained above, logic programs have been easily encoded into functional logic languages. However, a more natural operational semantics approach is possible in

Maude and, in [20], we specified a simple logic programming language illustrating various (logical) computation features. In this paper, we have considered how some extensions of logic programming, such as the ones discussed in [54,18], can easily be handled in Maude.

Many logic programming features have not yet been addressed in Maude. They seem very attractive topics for future research, including:

- Logic programming languages are well-known for efficient *indexing*. Maude provides very efficient matching, unification, rewriting, and narrowing algorithms but there is room for improvement.
- Exploring Or-Parallelism and And-parallelism. Parallel definitions of Prolog have been extremely useful in practice and much work in this direction could be done in Maude using meta-interpreters.
- Effective exploration mechanisms in Prolog have no equivalent symbolic feature in Maude, just the rewriting strategy language or the metalevel. A narrowing strategy language would be very useful in the future.
- Negation as symbolic failure is a fundamental feature of logic programming which, unfortunately, has not been very much studied in Maude.

Acknowledgements I would like to thank the ALP newsletter organizers for giving me the opportunity of presenting the developments of unification and narrowing in Maude in the context of the celebration of 50 years of logic programming. The ideas I have presented here are based on joint work with many colleagues I would like to mention. The effort done by the Maude Team is part of a prolonged and exciting interest on symbolic capabilities at different levels. Steven Eker has done an extremely good job of providing high-performance algorithms. Folding variant narrowing has been the basis for equational unification in Maude and is joint work with Jose Meseguer and Ralf Sasse, from ETH Zurich. Folding narrowing with rules is the basis of the Maude's Symbolic LTL Model Checker developed in joint work with Jose Meseguer and Kyungmin Bae, from KAIST. Both have been critical for the Maude-NPA protocol analyzer tool developed in a very productive joint work with Catherine Meadows and Jose Meseguer, as well as the different Ph.D. students we had. Many other applications of unification and narrowing in Maude have been developed during these years, for instance the work done in Valencia on anti-unification, homeomorphic embedding and partial evaluation of Maude programs with María Alpuente, Angel Cuenca-Ortega and Julia Sapiña but also with Jose Meseguer and Demis Ballis, from Udine.

References

1. Alpuente, M., Ballis, D., Frechina, F., Sapiña, J.: Debugging Maude Programs via Runtime Assertion Checking and Trace Slicing. Journal of Logical and Algebraic Methods in Programming **85**, 707–736 (2016)
2. Alpuente, M., Ballis, D., Escobar, S., Sapiña, J.: Optimization of rewrite theories by equational partial evaluation. J. Log. Algebraic Methods Program. **124**, 100729 (2022). https://doi.org/10.1016/j.jlamp.2021.100729, `https://doi.org/10.1016/j.jlamp.2021.100729`

3. Alpuente, M., Ballis, D., Falaschi, M.: Transformation and debugging of functional logic programs. In: Dovier, A., Pontelli, E. (eds.) 25 Years GULP. Lecture Notes in Computer Science, vol. 6125, pp. 271–299. Springer (2010)
4. Alpuente, M., Ballis, D., Sapiña, J.: Static Correction of Maude Programs with Assertions. Journal of Systems and Software **153**, 64–85 (2019)
5. Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Partial evaluation of order-sorted equational programs modulo axioms. In: Hermenegildo, M.V., López-García, P. (eds.) Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10184, pp. 3–20. Springer (2016). https://doi.org/10.1007/978-3-319-63139-4_1, `https://doi.org/10.1007/978-3-319-63139-4_1`
6. Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: A partial evaluation framework for order-sorted equational programs modulo axioms. J. Log. Algebraic Methods Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100501, `https://doi.org/10.1016/j.jlamp.2019.100501`
7. Antoy, S., Hanus, M.: Functional logic programming. Commun. ACM **53**(4), 74–85 (2010)
8. Aparicio-Sánchez, D., Escobar, S., Gutiérrez, R., Sapiña, J.: An optimizing protocol transformation for constructor finite variant theories in Maude-NPA. In: Chen, L., Li, N., Liang, K., Schneider, S.A. (eds.) Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12309, pp. 230–250. Springer (2020). https://doi.org/10.1007/978-3-030-59013-0_12, `https://doi.org/10.1007/978-3-030-59013-0_12`
9. Aparicio-Sánchez, D., Escobar, S., Meadows, C.A., Meseguer, J., Sapiña, J.: Protocol analysis with time. In: Bhargavan, K., Oswald, E., Prabhakaran, M. (eds.) Progress in Cryptology - INDOCRYPT 2020 - 21st International Conference on Cryptology in India, Bangalore, India, December 13-16, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12578, pp. 128–150. Springer (2020). https://doi.org/10.1007/978-3-030-65277-7_7, `https://doi.org/10.1007/978-3-030-65277-7_7`
10. Aparicio-Sánchez, D., Escobar, S., Meadows, C.A., Meseguer, J., Sapiña, J.: Protocol analysis with time and space. In: Dougherty, D., Meseguer, J., Mödersheim, S.A., Rowe, P.D. (eds.) Protocols, Strands, and Logic - Essays Dedicated to Joshua Guttman on the Occasion of his 66.66th Birthday. Lecture Notes in Computer Science, vol. 13066, pp. 22–49. Springer (2021). https://doi.org/10.1007/978-3-030-91631-2_2, `https://doi.org/10.1007/978-3-030-91631-2_2`
11. Arts, T., Zantema, H.: Termination of logic programs using semantic unification. In: Proietti, M. (ed.) Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings. Lecture Notes in Computer Science, vol. 1048, pp. 219–233. Springer (1996)
12. Bae, K., Escobar, S., Meseguer, J.: Abstract logical model checking of infinite-state systems using narrowing. In: van Raamsdonk, F. (ed.) 24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands. LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013). https://doi.org/10.4230/LIPIcs.RTA.2013.81, `http://dx.doi.org/10.4230/LIPIcs.RTA.2013.81`

13. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Sci. Comput. Program. **178**, 20–42 (2019). https://doi.org/10.1016/j.scico.2019.03.006, `https://doi.org/10.1016/j.scico.2019.03.006`
14. Chadha, R., Cheval, V., Ciobâcă, Ş., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. ACM Trans. Comput. Log. **17**(4), 23 (2016). https://doi.org/10.1145/2926715, `https://doi.org/10.1145/2926715`
15. Chen, X., Rosu, G.: The K vision for the future of programming language design and analysis. In: Bartocci, E., Falcone, Y., Leucker, M. (eds.) Formal Methods in Outer Space - Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 13065, pp. 3–9. Springer (2021). https://doi.org/10.1007/978-3-030-87348-6_1, `https://doi.org/10.1007/978-3-030-87348-6_1`
16. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude Manual (Version 3.2.1). Tech. rep., SRI International Computer Science Laboratory (2022), available at: `http://maude.cs.illinois.edu`
17. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3467, pp. 294–307. Springer (2005)
18. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained horn clauses for program verification. Theory Pract. Log. Program. **22**(6), 974–1042 (2022). https://doi.org/10.1017/S1471068421000211, `https://doi.org/10.1017/S1471068421000211`
19. Dershowitz, N.: Goal solving as operational semantics. In: International Logic Programming Symposium, Portland, OR. pp. 3–17. MIT Press (December 1995)
20. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Programming and Symbolic Computation in Maude. Journal of Logical and Algebraic Methods in Programming **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497
21. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.: Built-in Variant Generation and Unification, and their Applications in Maude 2.7. In: Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016). Lecture Notes in Computer Science, vol. 9706, pp. 183–192. Springer (2016)
22. Durán, F., Meseguer, J.: On the Church-Rosser and Coherence Properties of Conditional Order-sorted Rewrite Theories. The Journal of Logic and Algebraic Programming **81**(7–8), 816–850 (2012). https://doi.org/10.1016/j.jlap.2011.12.004
23. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Equational unification and matching, and symbolic reachability analysis in maude 3.2 (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13385, pp. 529–540. Springer (2022). https://doi.org/10.1007/978-3-031-10769-6_31, `https://doi.org/10.1007/978-3-031-10769-6_31`
24. Eker, S.: Associative unification in maude. J. Log. Algebraic Methods Program. **126**, 100747 (2022). https://doi.org/10.1016/j.jlamp.2021.100747, `https://doi.org/10.1016/j.jlamp.2021.100747`

25. Escobar, S.: Functional Logic Programming in Maude. In: Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi (SAS 2014). Lecture Notes in Computer Science, vol. 8373, pp. 315–336. Springer (2014)
26. Escobar, S., Kapur, D., Lynch, C., Meadows, C., Meseguer, J., Narendran, P., Sasse, R.: Protocol Analysis in Maude-NPA using Unification modulo Homomorphic Encryption. In: Proceedings of the 13th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2011). pp. 65–76. Association for Computing Machinery (2011)
27. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In: Foundations of Security Analysis and Design V (FOSAD 2007/2008/2009 Tutorial Lectures). Lecture Notes in Computer Science, vol. 5705, pp. 1–50. Springer (2009). https://doi.org/10.1007/978-3-642-03829-7_1
28. Escobar, S., Sasse, R., Meseguer, J.: Folding Variant Narrowing and Optimal Variant Termination. The Journal of Logic and Algebraic Programming **81**(7–8), 898–928 (2012). https://doi.org/10.1016/j.jlap.2012.01.002
29. Escobar, S.: Refining weakly outermost-needed rewriting and narrowing. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden. pp. 113–123. ACM (2003). https://doi.org/10.1145/888251.888263, https://doi.org/10.1145/888251.888263
30. Escobar, S.: Implementing natural rewriting and narrowing efficiently. In: Kameyama, Y., Stuckey, P.J. (eds.) Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2998, pp. 147–162. Springer (2004). https://doi.org/10.1007/978-3-540-24754-8_12, https://doi.org/10.1007/978-3-540-24754-8_12
31. Escobar, S.: Multi-paradigm programming in maude. In: Rusu, V. (ed.) Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11152, pp. 26–44. Springer (2018). https://doi.org/10.1007/978-3-319-99840-4_2, https://doi.org/10.1007/978-3-319-99840-4_2
32. Escobar, S., Meseguer, J.: Symbolic model checking of infinite-state systems using narrowing. In: Baader, F. (ed.) Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4533, pp. 153–168. Springer (2007)
33. Escobar, S., Meseguer, J., Thati, P.: Natural rewriting for general term rewriting systems. In: Etalle, S. (ed.) Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3573, pp. 101–116. Springer (2004). https://doi.org/10.1007/11506676_7, https://doi.org/10.1007/11506676_7
34. Escobar, S., Meseguer, J., Thati, P.: Natural narrowing for general term rewriting systems. In: Giesl, J. (ed.) Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3467, pp. 279–293. Springer (2005). https://doi.org/10.1007/978-3-540-32033-3_21, https://doi.org/10.1007/978-3-540-32033-3_21

35. Fay, M.: First Order Unification in an Equational Theory. In: Proceedings of the 4th International Conference on Automated Deduction (CADE 1979). pp. 161–167. Academic Press, Inc. (1979)
36. Goguen, J.A., Meseguer, J.: Equality, Types, Modules, and (why not?) Generics for Logic Programming. The Journal of Logic Programming **1**(2), 179–210 (1984)
37. Goguen, J.A., Meseguer, J.: Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In: Agha, G., Wegner, P., Yonezawa, A. (eds.) Research Directions in Object-Oriented Programming, pp. 417–478. The MIT Press (1987)
38. Goguen, J., Meseguer, J.: Eqlog: Equality, types and generic modules for logic programming. In: DeGroot, D., Lindstrom, G. (eds.) Logic Programming, Functions, Relations and Equations, pp. 295–363. Prentice-Hall (1986)
39. González-Burgueño, A., Aparicio-Sánchez, D., Escobar, S., Meadows, C.A., Meseguer, J.: Formal verification of the yubikey and yubihsm apis in Maude-NPA. In: Barthe, G., Sutcliffe, G., Veanes, M. (eds.) LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018. EPiC Series in Computing, vol. 57, pp. 400–417. EasyChair (2018). https://doi.org/10.29007/c4xk, `https://doi.org/10.29007/c4xk`
40. Gutiérrez, R., Lucas, S.: mu-term: Verify termination properties automatically (system description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12167, pp. 436–447. Springer (2020). https://doi.org/10.1007/978-3-030-51054-1_28, `https://doi.org/10.1007/978-3-030-51054-1_28`
41. Hanus, M.: The Integration of Functions into Logic Programming: From Theory to Practice. The Journal of Logic Programming **19/20**, 583–628 (1994)
42. Hanus, M.: Integration of Declarative Paradigms: Benefits and Challenges. ACM SIGPLAN Notices **32**(1), 77–79 (1997)
43. Hanus, M.: Functional Logic Programming: From Theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. Lecture Notes in Computer Science, vol. 7797, pp. 123–168. Springer (2013)
44. Hanus, M.: Curry: An integrated functional logic language (vers. 0.9.0) (2016), Available at: `http://www.curry-lang.org`
45. Hanus, M.: The integration of functions into logic programming: From theory to practice. Journal of Logic Programming **19&20**, 583–628 (1994)
46. Hanus, M.: Analysis of residuating logic programs. J. Log. Program. **24**(3), 219–245 (1995). https://doi.org/10.1016/0743-1066(94)00105-F, `https://doi.org/10.1016/0743-1066(94)00105-F`
47. Hanus, M.: Multi-paradigm declarative languages. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 45–75. Springer (2007)
48. Hanus, M.: Multiparadigm languages. In: Gonzalez, T.F., Diaz-Herrera, J., Tucker, A. (eds.) Computing Handbook, Third Edition: Computer Science and Software Engineering, pp. 66: 1–17. CRC Press (2014)
49. Hanus, M.: From logic to functional logic programs. Theory Pract. Log. Program. **22**(4), 538–554 (2022). https://doi.org/10.1017/S1471068422000187, `https://doi.org/10.1017/S1471068422000187`
50. Hermenegildo, M.V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J.F., Puebla, G.: An overview of ciao and its design philosophy. Theory Pract. Log. Program. **12**(1-2), 219–252 (2012). https://doi.org/10.1017/S1471068411000457, `https://doi.org/10.1017/S1471068411000457`

51. Hullot, J.M.: Compilation de Formes Canoniques dans les Théories Equationnelles. Ph.D. thesis, Université de Paris-Sud (1980)
52. Jaffar, J., Lassez, J.: Constraint logic programming. In: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987. pp. 111–119. ACM Press (1987). https://doi.org/10.1145/41625.41635, `https://doi.org/10.1145/41625.41635`
53. Jones, S.L.P.: Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. Journal of Functional Programming **2**(2), 127–202 (1992). https://doi.org/10.1017/S0956796800000319
54. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S.: Fifty years of Prolog and beyond. Theory Pract. Log. Program. **22**(6), 776–858 (2022). https://doi.org/10.1017/S1471068422000102, `https://doi.org/10.1017/S1471068422000102`
55. Lee, J., Kim, S., Bae, K.: Bounded model checking of PLC ST programs using rewriting modulo SMT. In: Artho, C., Ölveczky, P.C. (eds.) Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2022, Auckland, New Zealand, 7 December 2022. pp. 56–67. ACM (2022). https://doi.org/10.1145/3563822.3568016, `https://doi.org/10.1145/3563822.3568016`
56. López-Rueda, R., Escobar, S.: Canonical narrowing for variant-based conditional rewrite theories. In: Riesco, A., Zhang, M. (eds.) Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13478, pp. 20–35. Springer (2022). https://doi.org/10.1007/978-3-031-17244-1_2, `https://doi.org/10.1007/978-3-031-17244-1_2`
57. López-Rueda, R., Escobar, S.: Canonical narrowing with irreducibility and SMT constraints as a generic symbolic protocol analysis method. In: Bae, K. (ed.) Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13252, pp. 45–64. Springer (2022). https://doi.org/10.1007/978-3-031-12441-9_3, `https://doi.org/10.1007/978-3-031-12441-9_3`
58. Lucas, S., Meseguer, J.: Normal Forms and Normal Theories in Conditional Rewriting. Journal of Logical and Algebraic Methods in Programming **85**, 67–97 (2016). https://doi.org/10.1016/j.jlamp.2015.06.001
59. Lucas, S.: Context-sensitive rewriting. ACM Comput. Surv. **53**(4), 78:1–78:36 (2021). https://doi.org/10.1145/3397677, `https://doi.org/10.1145/3397677`
60. Martí-Oliet, N., Meseguer, J.: Rewriting Logic: Roadmap and Bibliography. Theoretical Computer Science **285**(2), 121–154 (2002)
61. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013). Lecture Notes in Computer Science, vol. 8044, pp. 696–701. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_48
62. Meseguer, J.: Multiparadigm logic programming. In: Kirchner, H., Levi, G. (eds.) Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2–4, 1992, Proceedings. Lecture Notes in Computer Science, vol. 632, pp. 158–200. Springer (1992)

63. Meseguer, J.: From OBJ to Maude and beyond. In: Futatsugi, K., Jouannaud, J., Meseguer, J. (eds.) Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 4060, pp. 252–280. Springer (2006). https://doi.org/10.1007/11780274_14, `http://dx.doi.org/10.1007/11780274_14`
64. Meseguer, J.: Strict coherence of conditional rewriting modulo axioms. Theor. Comput. Sci. **672**, 1–35 (2017). https://doi.org/10.1016/j.tcs.2016.12.026, `https://doi.org/10.1016/j.tcs.2016.12.026`
65. Meseguer, J.: Symbolic reasoning methods in rewriting logic and Maude. In: Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018, Bogota, Colombia, July 24-27, 2018, Proceedings. Lecture Notes in Computer Science, Springer (2018), to appear
66. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018). https://doi.org/10.1016/j.scico.2017.09.001, `https://doi.org/10.1016/j.scico.2017.09.001`
67. Meseguer, J.: Symbolic computation in maude: Some tapas. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12561, pp. 3–36. Springer (2020). https://doi.org/10.1007/978-3-030-68446-4_1, `https://doi.org/10.1007/978-3-030-68446-4_1`
68. 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)
69. Minsky, Y.: Ocaml for the masses: Why the next language you learn should be functional. Queue **9**(9), 40–49 (sep 2011). https://doi.org/10.1145/2030256.2038036, `https://doi.org/10.1145/2030256.2038036`
70. Nigam, V., Talcott, C.L.: Automating safety proofs about cyber-physical systems using rewriting modulo SMT. In: Bae, K. (ed.) Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13252, pp. 212–229. Springer (2022). https://doi.org/10.1007/978-3-031-12441-9_11, `https://doi.org/10.1007/978-3-031-12441-9_11`
71. Reddy, U.S.: Narrowing as the operational semantics of functional languages. In: Proceedings of the 1985 Second Symposium on Logic Programming, Boston, Massachusetts, July 15-18, 1985. pp. 138–151. IEEE Computer Society Press (1985)
72. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. J. Log. Algebraic Methods Program. **86**(1), 269–297 (2017). https://doi.org/10.1016/j.jlamp.2016.10.001, `https://doi.org/10.1016/j.jlamp.2016.10.001`
73. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Strategies, model checking and branching-time properties in maude. J. Log. Algebraic Methods Program. **123**, 100700 (2021). https://doi.org/10.1016/j.jlamp.2021.100700, `https://doi.org/10.1016/j.jlamp.2021.100700`
74. Slagle, J.R.: Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. Journal of the ACM **21**(4), 622–642 (1974)
75. Tushkanova, E., Giorgetti, A., Ringeissen, C., Kouchnarenko, O.: A rule-based system for automatic decidability and combinability. Science of Computer Programming **99**, 3–23 (2015). https://doi.org/10.1016/j.scico.2014.02.005, `http://dx.doi.org/10.1016/j.scico.2014.02.005`