Symbolic Computation in Maude: Some Tapas

José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Abstract. Programming in Maude is executable mathematical modeling. Your mathematical model *is* the code you execute. Both deterministic systems, specified equationally as so-called *functional modules* and concurrent ones, specified in rewriting logic as *system modules*, are mathematically modeled and programmed this way. But rewriting logic is also a *logical framework* in which many different logics can be naturally represented. And one would like not only to execute these models, but to *reason* about them at a high level. For this, *symbolic methods* that can automate much of the reasoning are crucial. Many of them are actually supported by Maude itself or by some of its tools. These methods are very general: they apply not just to Maude, but to many other logics, languages and tools. This paper presents some *tapas* about these Maude-based symbolic methods in an informal way to make it easy for many other people to learn about, and benefit from, them.

1 Introduction

1.1 What is Maude?

Maude is a high-performance declarative language whose modules are *theories in rewriting logic*, a simple, yet expressive, *computational logic* to specify and program *concurrent systems* as rewrite theories. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E \cup B, R)$ where:

- Σ specifies a *signature* of typed *function symbols*.
- $(\Sigma, E \cup B)$ is an *equational theory* specifying the concurrent system's *states* as elements of the *algebraic data type* (initial algebra) $T_{\Sigma/E \cup B}$ defined by $(\Sigma, E \cup B)$.
- R are *rewrite rules* specifying the system's *local atomic transitions*.
- *Concurrent Computation* = *Deduction* in $\mathcal{R}$ = *Concurrent Rewriting* in $\mathcal{R}$.

In Maude, a rewrite theory $\mathcal{R}$ named `FOO` is specified —with mostly self-explanatory syntax— as a so-called *system module* of the form: `mod FOO is` $(\Sigma, E \cup B, R)$ `endm`.

But, since when $R = \emptyset$, $\mathcal{R} = (\Sigma, E \cup B, R)$ becomes just an *equational theory*, Maude has a *functional sublanguage* of so-called *functional modules*. A functional module `BAR` is specified as follows: `fmod BAR is` $(\Sigma, E \cup B)$ `endfm`, where:

- $B \subseteq \{A, C, U\}$ is any combination of *associativity* (A) and/or *commutativity* (C) and/or *identity* (U) axioms, specified with the corresponding `assoc`, `comm`, and `id:` keywords, and
- the equations E, when used as left-to-right simplification rules, are *convergent*, i.e., Church-Rosser and terminating,[1] modulo the axioms B.

We make the exact same assumptions about B and E for a system module `mod FOO is` $(\Sigma, E \cup B, R)$ `endm`. What this intuitively means is that the states of the concurrent system so specified enjoy *structural axioms* B, and can also have *state-updating functions* computable by equational left-to-right simplification with the equations E modulo B.

[1] Termination can of course be dropped for some applications: the lambda calculus or a deterministic Turing machine can be easily specified as functional modules in Maude.

1.2 Symbolic Computation in Maude

Since all computation in Maude is performed by *logical deduction* in equational logic and/or rewriting logic, talking about *symbolic computation* seems tautological. But it isn't. The point is that the usual computations in a functional or system module involve elements of an algebraic data type $T_{\Sigma/E\cup B}$, which are represented as *ground terms* (terms without variables) in the syntax of Σ. But Maude supports many useful computations involving *terms with variables*. For example, for u and v terms with variables among the $x_1,...,x_n$, solving the so-called *$E \cup B$-unification problem* $u(x_1,...,x_n) =? v(x_1,...,x_n)$ means answering the question of whether the *constraint* $u(x_1,...,x_n) = v(x_1,...,x_n)$ is *satisfiable* in the algebraic data type $T_{\Sigma/E\cup B}$ for some instantiation of the variables $x_1,...,x_n$. So, roughly speaking, problems involving logical variables and their solutions are those I shall describe as symbolic computation problems. Maude, either directly or through Maude-based tools, supports the following symbolic computation features:

1. B-**Unification** (modulo any $B \subseteq \{A,C,U\}$),
2. B-**Generalization** (modulo any $B \subseteq \{A,C,U\}$),
3. E,B-**Variants** of a term t in a *convergent* $(\Sigma, E \cup B)$, which is *finitary* iff $(\Sigma, E \cup B)$ has the *finite variant property* (FVP), in the sense explained in Section 4,
4. $E \cup B$-**Unification** for any *convergent* $(\Sigma, E \cup B)$, which is *finitary* iff $(\Sigma, E \cup B)$ is FVP,
5. **Domain-Specific SMT-Solving**, thanks to CVC4 [19] and Yices [74] interfaces,
6. **Theory-Generic SMT-Solving** for FVP theories $(\Sigma, E \cup B)$ under natural requirements about their constructors,
7. **Symbolic Reachability Analysis** of any system module **mod** $(\Sigma, E \cup B, R)$ **endm** with $(\Sigma, E \cup B)$ FVP,
8. B-**Homeomorphic Embedding** (modulo any $B \subseteq \{A,C\}$).

In this paper I will focus on features (1), (3)–(4), and (6)–(7) in the above list. For generalization modulo B —which is dual to unification and is also called "anti-unification"— please see [4, 2]. Homeomorphic embedding is a very useful relation for termination criteria in various symbolic analyses. It has been generalized for the first time to work in an order-sorted setting and modulo combinations of associativity and commutativity axioms, with new efficient algorithms, in [1]. Both generalization and homeomorphic embedding modulo axioms are crucial components of the variant-based partial evaluation (PE) approach for Maude functional modules presented in [3].

1.3 Tapas and Paper Napkins

To explain the symbolic features (1), (3)–(4), and (6)–(7) requires explaining some basic technical ideas that convey the precise meaning of such features. But this runs the risk of getting us bogged down in technicalities. How shall we proceed? I propose that we use our imagination a little: think of this paper as an informal conversation that you, dear reader, and I are having in a *Tapas Bar*, as we share some pleasant tapas and wash them down with some good *Rioja*. The bar's setting is informal: instead of sitting at a formal table, we sit at a small wooden table where there is a stack of small paper napkins. Tapas are now gradually making their appearance at two levels: each time our waiter brings us the next tapas serving, there are also some *Maude tapas* that I explain to you by scribbling on the paper napkins in the stack. The Maude tapas have to be *small*, since these are cocktail napkins. I have also brought my laptop to run a few examples; but the main action is our conversation, scribbling on paper napkins. Of course, a few technicalities have to be glossed over: I just give you the main intuitions; but I promise to email you some material to fill in those details later. This is what we are going to do here. In this paper, that more precise technical background can be found in Section 7 and in the list of references; but let us disregard them for now.

2 First Tapas Serving: Rewriting Modulo Axioms *B*

I have always claimed and felt that Maude, unlike other programming languages, *can be explained on a paper napkin* to somebody with no prior acquaintance with computing. Here is the example I would write on such a napkin:

```
fmod NATURAL is
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .

vars N M : Nat .

eq N + 0 = N .
eq N + s(M) = s(N + M) .
endfm
```

This module, defining natural number addition in Peano notation, does of course fit the general pattern `fmod BAR is` $(\Sigma, E \cup B)$ `endfm`, where here the module's name `BAR` is `NATURAL`, the typed signature Σ has a single type (called a *sort* in Maude), which we have *chosen* to call `Nat`, a constant `0` and two function symbols: `s` and `_+_`, where the underbars indicate argument positions, and where the `ctor` attribute is declared for `0` and `s` as *data constructors* to distinguish them from the *defined function* `_+_`, which is *defined* by the two equations E. In this case there are no atributes B, although, if we wished, we could have declared `_+_` with the `assoc` and `comm` keywords as an associative and commutative operator.

How do we compute with this module? By *simplifying* any arithmetic expression to its *result* as a *data value*, i.e., either to 0 or to $s^n(0)$ for some $n \geqslant 1$, using the two equations E to perform *left-to-right* replacement of *equals for equals* in the usual way this is done in algebraic simplification. This process is called *term rewriting*; and the result of thus simplifying an expression is called its *normal form*. Let us see (in another paper napkin) how this process reduces adding 2 plus 2, i.e., the arithmetic expression `s(s(0)) + s(s(0))` to 4, i.e., the data value `s(s(s(s(0))))`. For this, it is useful to add some simple notation to indicate *where* in an expression a simplification is applied. I will use the notation $t[u]$ to indicate that we are focusing on the subexpression u of the expression, or term, t. The process in this notation is as follows:

$$[s(s(0))+s(s(0))] \to s([s(s(0))+s(0)]) \to s(s([s(s(0))+0])) \to s(s(s(s(0))))$$

where we have applied the second equation in the first two steps, and the first equation in the last step, to corresponding *instances* by some *matching substitution* instantiating the equation's variables to the term or subterm to be simplified. For example, in the second step, the variables N and M have been instantiated by the substitution $\theta = \{N \mapsto s(s(0)), M \mapsto 0\}$, so that the subterm we focus on, $s(s(0))+s(0)$, becomes an instance of the pattern term $N+s(M)$ in the second equation's lefthand side, and is *replaced* in this step by the corresponding instance of the righthand side $s(N+M)$. We can summarize this (focused) step in the following notation:

$$s(s(0))+s(0) \equiv (s(N)+M)\theta \to s(N+M)\theta \equiv s(s(s(0))+0)$$

where $\equiv$ denotes syntactic equality, and $t\theta$ denotes the result of instantiating a pattern term, i.e., a term with variables t, by a substitution θ.

But Maude's functional modules do support this kind of algebraic simplification *modulo* structural axioms B. Let us illustrate this case with a simple example (it fits on another paper napkin) of a data type of sets:

```
fmod SET is
sort Set .
ops mt a b c d e f g : -> Set [ctor] .
op _U_ : Set Set -> Set [ctor assoc comm] .  *** union
vars S S' : Set .

eq S U mt = S [variant] .                     *** identity
eq S U S = S [variant] .                      *** idempotency
eq S U S U S' = S U S' [variant] .            *** idempotency
endfm
```

Its constants are `a b c d e f g` and the empty set constant `mt`. There is also a union operator, for which we have *chosen*[2] the syntax `_U_`, which has been declared associative (A) and commutative (C) by the `assoc` and `comm` attributes. Note that in this module all constants and `_U_` are *data constructors*. Set union is *defined* by the three equations (the third one follows from the second: it is added for technical reasons) of `mt` as identity element for set union, and set idempotency. Disregard for the moment the `[variant]` attribute in the equations: it will become clear in Section 4. Let us see an example of how we compute in this module *modulo AC*.

$$mt \cup [a \cup c \cup b \cup a \cup b] \to [mt \cup a \cup b \cup c] \to a \cup b \cup c$$

where we have used the third equation in the first step, and the first equation in the second step. Note that, because of associativity, we, as well as the Maude parser, can dispense with parentheses. The most interesting step is the first one, which uses the substitution $\theta = \{S \mapsto (a \cup b), S' \mapsto c\}$. This step can be applied because:

$$(S \cup S \cup S')\theta \equiv (a \cup b) \cup (a \cup b) \cup c =_{AC} a \cup c \cup b \cup a \cup b.$$

Since, thanks to the AC axioms, reordering and parentheses do not matter, the crucial point is that the subterm $a \cup c \cup b \cup a \cup b$ *is* an *instance* of the lefthand side pattern $S \cup S \cup S'$ *modulo AC*. For the same reason, the fact that *mt* appears on the *left* of the expression instead than on the right is no obstacle for applying the first equation in the second step modulo AC.

It can be easily checked that the equations in `NATURAL`, resp. `SET`, are *convergent*, and therefore the *normal forms* of, for example, $s(s(0)) + s(s(0))$, resp. $mt \cup a \cup c \cup b \cup a \cup b$, namely, $s(s(s(s(0))))$, resp. $a \cup b \cup c$, are *unique* modulo B, regardless of the order in which the equations are applied to the original term. For example, $b \cup c \cup a$ *is* the *same* normal form as $a \cup b \cup c$ modulo AC. The Maude command computing a term's normal form is the `reduce` command.

A little notation does not hurt anybody. The process of performing *one step* of *rewriting* a term t (focusing on *some* subterm) using one of the equations in E *modulo* the axioms B to obtain a term t' is called *E,B-rewriting*, and is denoted $t \to_{E,B} t'$. Likewise, $t \to^*_{E,B} t'$ denotes performing zero, one or more steps of E,B-rewriting. The special case when $B = \emptyset$ is called E-rewriting, and then we use the notation $t \to_E t'$ and $t \to^*_E t'$. The *E,B-normal form* of term t (unique up to B-equality assuming E convergent) is denoted $t!_{E,B}$, resp. $t!_E$ when $B = \emptyset$.

[2] In Maude, all syntax for sort and operator names is *user-definable*.

3 Second Tapas Serving: Unification and Narrowing Modulo *B*

As already mentioned, solving a *B-unification problem* $u(x_1,...,x_n) =?\, v(x_1,...,x_n)$ means answering the question of whether the *constraint* $u(x_1,...,x_n) = v(x_1,...,x_n)$ is *satisfiable* in the algebraic data type $T_{\Sigma/B}$, where terms are identified modulo the axioms B, such as any combination of A and/or C and/or U axioms. The case $B = \varnothing$ is called *syntactic unification*. It is well-known from the Prolog language, where the analog of the data type T_Σ is the so-called *Herbrand model*, which extends T_Σ by adding predicate symbols. Maude supports unification modulo B in any module where the axioms B have been declared. Furthermore, this B-unification is *order-sorted*, i.e., it is carried out with variables which can have different *sorts*, where some of them can be *subsorts* of other sorts. In particular, since for the module `NATURAL` we have $B = \varnothing$, we can perform syntactic unification in it with Maude's `unify` command.

Since the syntactic case is well-known, and we will revisit it soon, let us focus instead on the more interesting case of the `SET` module, where we can perform AC-unification. What does this mean? Except for the fact that we are not dealing with the equation making `mt` the identity for `_U_`, this means that we can solve *multiset equations*, as opposed to solving *set equations* (but, please, be patient: we will also solve set equations in the next serving of tapas). For example, we may wish to solve the multiset equation: $a \cup a \cup b \cup S = a \cup c \cup S'$, that is, seek substitutions θ such that $(a \cup a \cup b \cup S)\theta =_{AC} (a \cup c \cup S')\theta$, i.e., both side instances yield the same multiset. We can do so in Maude by giving the command:

```
Maude> unify in SET : a U a U b U S =? a U c U S' .

Unifier 1
S --> c U #1:Set
S' --> a U b U #1:Set

Unifier 2
S --> c
S' --> a U b
```

where the second solution is the most obvious, and the first solution allows adding to the multiset $a \cup a \cup b \cup c$ obtained by the second solution an extra multiset denoted by the extra variable `#1:Set`.

Maude supports unification modulo *any* possible combinations of A, C, and U axioms in B; also when some axioms in B are declared associative but are *not* commutative. This is noteworthy, since it is well-known that the number of A-unifiers (or AU-unifiers) of a problem can be infinite. For example, if a is a constant and $\_\cdot\_$ is associative, then the equation $a \cdot x = x \cdot a$ has the infinite set of solutions: $\{\{x \mapsto a^n\} \mid n \geqslant 1\}$. When some operators are A or AU only, Maude's implementation of B-unification takes the following pragmatic approach: (i) the unification algorithm is designed to *favor the cases where the number of* A or AU-unifiers is known to be *finite*; and (ii) in all other cases, it searches for solutions in a complete manner, but *within a bound*, so that: (a) if *all* solutions are found before reaching the bound, it just returns them, but (b) if the bound is reached without the certainty of having found all solutions, *the solutions already found are returned* with a *warning* that the set of solutions may be incomplete. The good news is that, for a good number of applications —for example in the symbolic analysis of various cryptographic protocols involving associativity axioms— such warnings are never encountered, i.e., the corresponding analyses are then, luckily, *complete*.

Narrowing. This is just technical jargon for *symbolic execution*, in the usual sense one would expect: executing a program, not on concrete inputs, but on "symbolic" inputs specified by variables [38, 40]. In our case, a Maude functional module and a term with variables in its syntax. For example, in our `NATURAL` functional module for natural number addition, the symbolic expression $x+y$ *cannot* be evaluated in the standard sense: it is *already* in normal form, since *no equation* in `NATURAL` can be used to further simplify it. However, it *can* be executed *symbolically*. What does this mean? It means answering the following question:

> *Are there instances of $x+y$ that can be executed in the standard sense? And, if so, can we systematically describe them and their results?*

The answer, for any equational theory $(\Sigma, E \cup B)$ where the equations E are *convergent* modulo the axioms B is an emphatic Yes! The method is very simple, and amounts to a *slight generalization* of the already-described E,B-rewriting relation $\rightarrow_{E,B}$ between terms, to the more general E,B-*narrowing* relation $\rightsquigarrow_{E,B}$ between terms. What is this generalization like? Very simple: we replace the process of B-*matching* a subterm u as a substitution instance of the lefthand side t of an equation $t = t'$ by one of B-*unifying* t and u, that is, of solving the equation $t =? u$ modulo B.

In which sense is this a *slight* generalization? In the precise sense that when u is a *ground term*, i.e., it has no variables, then B-unification *coincides* with B-matching. For example, the matching substitution $\theta = \{S \mapsto (a \cup b), S' \mapsto c\}$ by which we showed that $(S \cup S \cup S')\theta =_{AC} a \cup c \cup b \cup a \cup b$ is indeed an AC-*unifier* (not the only one) of the equality $(S \cup S \cup S') =? a \cup c \cup b \cup a \cup b$.

The crucial point, however, is that when the term u to be evaluated *does have variables*, B-unification is *strictly more general* than B-matching and makes *symbolic execution* possible: because we now view the variables of u as *logical variables* in the Prolog sense, which can be *instantiated*. Let us see how $x+y$ can be symbolically executed this way. In `NATURAL` we have two equations $E = \{N+0 = N, N+s(M) = s(N+M)\}$. Focusing on the entire term $x+y$ we get two corresponding unification problems $N+0 =? x+y$ and $N+s(M) =? x+y$ with respective unifiers $\theta_0 = \{N \mapsto x, y \mapsto 0\}$ and $\theta_1 = \{N \mapsto x, M \mapsto y', y \mapsto s(y')\}$. Applying these substitutions to the righthand sides of the equations we get the narrowing steps:

$$[x+y] \rightsquigarrow_E^{\theta_0} x \quad \textit{and} \quad [x+y] \rightsquigarrow_E^{\theta_1} s(x+y')$$

where we have indicated for each step the substitution used: θ_0, resp. θ_1. Narrowing *is never performed on variables*, so the first narrowing step cannot be continued. But the second can, focusing on the subterm $x+y'$, again in two ways, by the substitutions: $\theta'_0 = \{N \mapsto x, y' \mapsto 0\}$ and $\theta'_1 = \{N \mapsto x, M \mapsto y'', y' \mapsto s(y'')\}$, yielding narrowing steps:

$$s([x+y']) \rightsquigarrow_E^{\theta'_0} s(x) \quad \textit{and} \quad s([x+y']) \rightsquigarrow_E^{\theta'_1} s(s(x+y''))$$

And, obviously, since $s(x)$ cannot be unified with any lefthand side, it is only the second term (focusing on $x+y''$) that can be narrowed again, *in exactly the same way*, ad infinitum. We get this way what is called an (infinite) *narrowing tree* rooted at our original term $x+y$. But we could have started with *any* other term in the syntax of `NATURAL`. In the same way, but in this case performing unification modulo AC, the three equations E in the `SET` module define a narrowing relation $\rightsquigarrow_{E,AC}$ which performs *symbolic execution of set expressions*. Of course, we also have a reflexive-transitive closure $\rightsquigarrow^*_{E,AC}$, which, when annotated with a substitution, $\overset{\theta}{\rightsquigarrow}{}^*_{E,AC}$ makes explicit the *composed or "accumulated" substitution* $\theta = \theta_1 \cdots \theta_n$ for a length-n narrowing sequence.

Note the interesting fact that, although the equations E of a convergent theory, such as `NATURAL` or `SET`, are always *terminating*, the associated narrowing relation $\rightsquigarrow_{E,B}$ in general is not. When does it terminate? This is a topic that we can save for the next tapas serving.

4 Third Tapas Serving: Variants, and Unification Modulo $E \cup B$

Let us you, dear reader, **DR**, and I, **JM**, play a little *language game* à la Wittgenstein. **JM**: What is a variant? **DR**: I don't know what you are talking about. **JM**: I mean, what is a variant in the Comon-Delaune [18] sense? **DR**: I don't know: you tell me. **JM**: An answer to a question. **DR**: Which question? **JM**: What are the normal forms that a term t in a Maude functional module evaluates to? **DR**: But the answer to your question is trivial, since we have already seen that, since the module's equations E are assumed convergent modulo its axioms B, up to B-equality there is just *one answer*, namely, the unique normal form $t!_{E,B}$ of t, which is the answer provided by Maude's `reduce` command. **JM**: Sorry, what I really meant is: What are the normal forms that a term t *symbolically* evaluates to? Or, slightly more broadly: What are the normal forms of the *instances* of t by various *substitutions*? **DR**: Well, that sounds more interesting. Can you give me an example? **JM**: Why, of course! We have just *seen* an example! **DR**: Where? **JM**: In the last *paper napkin* I scribbled for you, where I sketched the *narrowing tree* for $x+y$. **DR**: What do you mean? **JM**: (1) A little reflection shows that, if we have a narrowing sequence: $t\overset{\theta}{\leadsto}{}^{*}_{E,B}u$, and u is normalized, then, by construction, $u =_B (t\theta)!_{E,B}$ and u is therefore a variant *in the exact sense I meant*. (2) But if you inspect the narrowing tree for $x+y$, all the terms in that tree are either of the form: $s^n(x)$, $n \geqslant 0$, or $s^n(x+y'^n)$, $n \geqslant 1$, which are all *in normal form*. So they are all *variants* of $x+y$ in the sense I just meant. **DR**: Ok, now I see your point. This looks interesting. Tell me more. **JM**: Of course, these terms are not *all* the variants of $x+y$. But they *cover* all the variants of $x+y$ as *instances*. For example, the substitution $\theta = \{x \mapsto s(0+x'), y \mapsto s(s(z))\}$ yields the variant: $((x+y)\theta)!_E = s(s(s(0+x')+z))$, which is itself an *instance* of the term $s(s(x+y''))$ in $x+y$'s narrowing tree. Therefore, —because of the so-called *lifting property of narrowing* (references in Section 7.2)— we can use a term's t narrowing tree to compute a *complete set of most general variants* of t by just selecting those narrowing paths in such a tree of the form $t\overset{\theta}{\leadsto}{}^{*}_{E,B}u$, where u is normalized. **A little more notation cannot hurt**. For technical reasons, we do not call such a u a variant of t. Instead, we formally define that variant as the *pair* (u,θ). This is because we might have a quite different (u',γ), with u' just a variable renaming of u, obtained by a completely different narrowing path $t\overset{\gamma}{\leadsto}{}^{*}_{E,B}u'$, and where γ itself might *not* be a variable renaming of θ. We shall see examples like this during this tapas serving.

The Finite Variant Property. Here are two closely-related, yet different, questions. Given a Maude functional module, say, `fmod BAR is` $(\Sigma, E \cup B)$ `endfm`, as always with E assumed *convergent* modulo B,

1. When is it the case that any term t in this module has a *finite*, complete set of most general variants —i.e., that, up to B-equality, any other variant of t is a substitution instance of one in this finite set? If this holds, we then say that $(\Sigma, E \cup B)$ has the *finite variant property* (FVP).
2. When does E,B-narrowing *terminate* for any term t in this module?

Since, as we have just seen, a complete set of variants of a term t can be computed by narrowing, *if* E,B-narrowing terminates for all inputs t, *then* $(\Sigma, E \cup B)$ is obviously FVP. But the converse does not hold in general: a term t may have a *finite* set of most general variants and yet have an *infinite* narrowing tree. Why? Because we should do something *smarter* than just generating t's narrowing tree. The problem we can easily face when generating t's narrowing tree is that, after a while, *if* we had looked carefully enough, *we would have seen it all*. That is, seen that any variant to be generated further down the (infinite!) tree is going to be an *instance* of one that we have already seen. But how can we *find that out*, since the tree is *infinite*? By using the *folding variant narrowing strategy* in [27]. This strategy has the useful property that: (1) $(\Sigma, E \cup B)$ is FVP iff (2) folding variant E,B-narrowing *terminates* for any input term t. Folding variant narrowing *computes* the desired finite set of most general variants of a term t when $(\Sigma,E\cup B)$ is FVP; and *in all cases* —i.e., for any convergent $(\Sigma,E\cup B)$— it computes a *complete* set of variants of t, which may of course be *infinite*. For example, `NATURAL` is *not* FVP. This is obvious from the fact that, for any two $n,k\geqslant 1$, the terms $s^{n}(x+y'^{n})$ and $s^{n+k}(x+y'^{n})$ have *disjoint* sets of instances.

But how does folding variant narrowing work? As its name suggests, by *folding*. That is, we do not generate a *tree*, but a *graph* in a breadth first way. But when we generate a new normalized node, we do not just add it to the graph: we first *check* to see if in the graph generated so far we already have another node of which this new one is an *instance* and, if so, we *fold* the new node into that most general instance. If at some depth all new generated nodes must be folded, then we have terminated with a finite graph that contains a set of most general variants of the input term t.

Folding variant narrowing has been implemented in Maude. The set of variants of a term t can be computed with Maude's `get variants` command. Since in general this set can be infinite, the user can provide a bound n to get the first n variants of a term t. But how can we *know* if a given $(\Sigma,E\cup B)$ is FVP? This property is *undecidable* [8]. However, as explained in [12], *if* $(\Sigma,E\cup B)$ *is* actually FVP, provided that B-unification is finitary,[3] we can find this out very easily in Maude by computing the variants of each term $f(x_1,...,x_n)$ for each function symbol f in Σ. For example, our `SET` example, which can easily be shown convergent, *is* FVP, since Maude provides the following answer:

```
Maude> get variants in SET : S U S' .

Variant 1
Set: #1:Set U #2:Set
S --> #1:Set
S' --> #2:Set

Variant 2
Set: %1:Set
S --> mt
S' --> %1:Set

Variant 3
Set: %1:Set
S --> %1:Set
S' --> mt

Variant 4
Set: %1:Set
S --> %1:Set
S' --> %1:Set

Variant 5
Set: %1:Set U %2:Set U %3:Set
S --> %1:Set U %2:Set
S' --> %1:Set U %3:Set

Variant 6
```

[3] As already mentioned, if B contains axioms of associativity without commutativity, B-unification will not be finitary. The FVP property has been studied for this more general case in [49].

```
Set: %1:Set U %2:Set
S --> %1:Set U %2:Set
S' --> %2:Set

Variant 7
Set: %1:Set U %2:Set
S --> %2:Set
S' --> %1:Set U %2:Set

No more variants.
```

which shows that `SET` is FVP. Note that, in general, a functional module's equational theory $(\Sigma,E\cup B)$ need not be FVP. In reality, what the `get variants` command for a term t provides is a very *space-efficient* way of describing the narrowing tree of a term t, not as a tree, but as a *graph with folding* storing only *normalized nodes*. In comparison with the tree description itself, this space efficiency is enormous in all cases; and in the FVP case it can reduce an *infinite* tree to a *finite* graph. Pragmatically, —particularly in the case of axioms such as AC where the number of unifiers of a unification problem can be huge and therefore the narrowing tree can have large degrees of branching— the difference between a term's narrowing tree and its narrowing graph with folding is one between a hopeless procedure that can be easily overwhelmed at very small tree depths and a practical procedure that can be used in many applications.

Constructor Variants. As we have seen in the `NATURAL` and `SET` modules, Maude supports the distinction between *constructor operators*, which build data and are specified with the `ctor` attribute, e.g., `0` and `s` in `NATURAL`, and the remaining *defined function symbols*, like `_+_` in `NATURAL`. This offers a very natural distinction at the level of variants: we call a variant (u,θ) of a term t a *constructor variant* iff u is a constructor term, that is, a term built using only constructor symbols and variables. Since in the `SET` module all symbols are constructor symbols, the above seven variants of the term `S U S'` are all constructor variants. Instead, in the already-described complete set of variants for the term $x+y$ in `NATURAL`, only the family of terms $\{s^n(x) \mid n \geqslant 0\}$ are constructor variants. This distinction between variants and constructor variants will prove useful in our next tapas serving.

Variant $E\cup B$-Unification. So far, we have only discussed Maude's algorithm for B-unification, with B any combination of A, C, and U axioms. Though very useful, this is also very limited. Assuming, as I will do throughout, that *all sorts are inhabited*, i.e., algebraic data types that do not have empty types/sorts, what B-unification really means is that we can answer satisfiability questions for constraints of the form: $\bigwedge_{1\leqslant i\leqslant n} u_i = v_i$ in algebraic data types of the form $T_{\Sigma/B}$. But, of course, what we would like to be able to do is to solve the same kind of constraints for *any* Maude functional module, under the assumptions that it is convergent and that its equations are unconditional. That is, to be able to solve the above constraints over algebraic data types of the form $T_{\Sigma/E\cup B}$. In other words, to perform $E\cup B$-unification. For example, we already saw that for $(\Sigma,E\cup AC)$ the equational theory of the `SET` module, AC-unification, i.e., solving equations in $T_{\Sigma/AC}$ essentially amounted to *multiset unification* —up to a minor quibbling about the empty set that could have been solved adding an extra U axiom. But what we really would like to perform is *set unification*, i.e., to solve constraints of the above form in the data type $T_{\Sigma/E\cup AC}$ of *sets*. Can we do this? The answer is Yes! Because we can *reduce* such a unification problem to one of *computing variants*. Let us see how. All we need to do[4] is to add to our functional module of choice a new sort `Pred` of predicates with constant `true`, and a new *equality predicate*. Let us illustrate this idea for the `SET` module, extended to the module:

```
fmod SET-EQ is protecting SET .
sort Pred .                        *** Predicates sort
```

[4] For simplicity, I treat the case of solving a single equation. The case of solving systems of equalities and disequalities can likewise be treated by adding a binary conjunction operator to `Pred` with identity `true`.

```
op true : -> Pred [ctor] .
op _=?_ : Set Set -> Pred [ctor] .  *** equality predicate
vars S S' : Set .

eq S =? S = true [variant] .        *** equality definition
endfm
```

It is easy to check that this module is also FVP. This is a general fact: the extension of an FVP theory $(\Sigma,E\cup B)$ to a theory $(\Sigma^{=?},E^{=?}\cup B)$ by adding an equality predicate $\_=?\_$ is always also FVP. This can be easily checked in this example by computing the variants of the term `S =? S'`. Recall that, using AC unification, we were able to answer the *multiset unification problem*: `a U a U b U S =? a U c U S'`. But what we would like to do is to solve the *set unification problem*: `a U a U b U S =? a U c U S'`. We can do so by computing variants in `SET-EQ` of the equality term `a U a U b U S =? a U c U S'`. Maude returns 88 such variants. But the only ones that interest us are those of the form: $(true,\theta)$, since those θ are the desired unifiers for this set unification problem. There are only 24 variants of the form $(true,\theta)$, which give us our desired family of set unifiers. Here are the first and the last of these:

```
Maude> get variants in SET-EQ : a U a U b U S =?  a U c U S' .
...

Variant 2
Pred: true
S --> c U %1:Set
S' --> b U %1:Set
...

Variant 88
Pred: true
S --> b U c
S' --> a U b U c
```

But *why* are these the unifiers of our set equation? **Never let a theorem that fits on a paper napkin go to waste!** Because, as explained in Section 7.2, for any convergent theory $(\Sigma,E\cup B)$ we have the *Church-Rosser Equivalence*: $t =_{E\cup B} t' \Leftrightarrow t!_{E,B} =_B t'!_{E,B}$. Therefore, a substitution θ solves an equation $u = ?v$ in $T_{\Sigma/E\cup B}$ iff $(u\theta)!_{E,B} =_B (v\theta)!_{E,B}$, i.e., iff $((u = ?v)\theta)!_{E^{=?},B} =_B true$. That is, iff θ is an *instance* of some γ in some variant of $u = ?v$ of the form $(true,\gamma)$. q.e.d. Note that this proof is much more general than: (i) solving equations for the `SET` module; (ii) solving equations for any FVP theory $(\Sigma,E\cup B)$; since (iii) it solves them for *any convergent* theory $(\Sigma,E\cup B)$. That is, this method provides a general $E\cup B$-unification procedure for *any* convergent theory $(\Sigma,E\cup B)$, which we call the *variant unification* procedure. However, the case when $(\Sigma,E\cup B)$ is FVP is noteworthy since, if B-unification is *finitary* (the case when any A axiom is also AC), then variant $E\cup B$-unification *is also finitary* and in fact a *satisfiability decision procedure*. That is, we can decide in a finite number of steps whether a constraint of the form $\bigwedge_{1\leqslant i\leqslant n} u_i = v_i$ is satisfiable in the algebraic data type $T_{\Sigma/E\cup B}$. For the same reason, we can also decide the satisfiability in $T_{\Sigma/E\cup B}$ of any *positive* (no negations) DNF formula of the form: $\bigvee_{1\leqslant i\leqslant n}\bigwedge_{1\leqslant i.j\leqslant n_i} u_{i.j} = v_{i.j}$. This suggests the question: What about satisfiability of *any* quantifier free (QF) formula in $T_{\Sigma/E\cup B}$? We will revisit this question in the next tapas serving.

$E\cup B$-unification is so important that, rather than solving a $E\cup B$-unification problem $u = ?v$ by computing the variants of the term $u = ?v$ in $(\Sigma^{=?},E^{=?}\cup B)$, which would yield other useless variants, Maude supports it *directly* in $(\Sigma,E\cup B)$, for systems of equations $\bigwedge_{1\leqslant i\leqslant n} u_i = v_i$, by the `variant unify` command. But since the set of $E\cup B$-unifiers computed this way often contains some unifiers that are less general than some other unifier in the set and are therefore *redundant*, Maude also supports a somewhat more expensive —yet quite practical for reducing the size of many symbolic search problems— command that filters out redundant $E \cup B$-unifiers, namely, the `filtered variant unify` command. For our example, it reduces the number of set unifiers from 24 to 9:

```
Maude> filtered variant unify in SET : a U b U c U S =? a U b U S' .

Unifier 1
S --> %1:Set
S' --> c U %1:Set

Unifier 2
S --> a U #1:Set
S' --> c U #1:Set

Unifier 3
S --> b U #1:Set
S' --> c U #1:Set

Unifier 4
S --> #1:Set
S' --> a U c U #1:Set

Unifier 5
S --> #1:Set
S' --> b U c U #1:Set

Unifier 6
S --> a U b U %1:Set
S' --> c U %1:Set

Unifier 7
S --> a U %1:Set
S' --> b U c U %1:Set

Unifier 8
S --> b U %1:Set
S' --> a U c U %1:Set

Unifier 9
S --> %1:Set
S' --> a U b U c U %1:Set

No more unifiers.
```

5 Fourth Tapas Serving: Variant Satisfiability

In computer science, decision procedures are used to automate reasoning about *data types*. In a conventional language, such data types may include integers, rational numbers, strings of characters, arrays, and so on. There is typically a finite collection of such data types used in a given programming language, which are often well supported by current SMT solvers. A theorem prover to verify programs in a conventional language can make very good use of such decision procedures to automate large portions of a program's proof of correctness. In Maude the situation is quite different. Why? Because in Maude algebraic data types are completely *user-definable*. That is, *any* functional module `fmod BAR is` $(\Sigma, E \cup B)$ `endfm` for *any*, finitely specifiable, convergent equational theory $(\Sigma, E \cup B)$ can be specified by a Maude user to define the algebraic data type $T_{\Sigma/E \cup B}$ of his/her choice. And, unlike the case of a conventional language, there is an *infinite* collection of such data types. Of course, for *some* specific Maude data types, for example integers or rationals, existing *domain-specific* decision procedures supported by an SMT solver may be available. But to automate reasoning about *arbitrary* Maude functional modules as much as possible, we need a new kind of SMT solving: what I call *theory-generic* decision procedures, which apply, not to a given data domain, but to an *infinite* class of *user-definable* data types. The generic decision procedure in question is called *variant satisfiability* [56], and is what this tapas serving is about.

The first piece of good news is that, for B any combination of A, C, and U axioms, where any A symbol f must also be C, satisfiability of QF formulas in the data type $T_{\Sigma/B}$ is *decidable* [56]. The million-dollar question is: How can we take advantage of this piece of good news to obtain a much more general *theory-generic* satisfiability decision procedure to help us reason about *any* algebraic data type $T_{\Sigma/E \cup B}$ defined by a Maude functional module `fmod BAR is` $(\Sigma, E \cup B)$ `endfm`? Of course, we know *a priori* that the class of algebraic data types $T_{\Sigma/E \cup B}$ for which we can hope to have decidable satisfiability, even if infinite, must have some restrictions: since just for the data type of natural numbers with addition and multiplication, that is, just by adding a multiplication operator $\_*\_$ and the equations $N*0 = 0, N*s(M) = N+(N*M)$ to our `NATURAL` module, Gödel's Incompleteness Theorem rears its head dashing all our decidable satisfiability hopes to the ground. So, one way to both rephrase the original question and advance towards an answer is to ask the more precise question:

> Given a Maude functional module `fmod BAR is` $(\Sigma, E \cup B)$ `endfm`, is there a *general method* by which we could seek, and find, a sublanguage of QF formulas, say, determined by a subsignature $\Sigma_1 \subseteq \Sigma$ such that satisfiability of QF Σ_1-formulas in $T_{\Sigma/E \cup B}$ is *decidable*?

What is promising about trying to answer this question is its *practical* character: hoping for decidable satisfiability of just *any* algebraic data type is both an act of self-delusion and a mark of ignorance. But hoping for a *subclass* of formulas enjoying decidable satisfiability is an eminently practical idea, which can help automate large parts of a program's proof of correctness effort.

The second piece of good news is that a general method answering the above question does indeed exist. It is based on the idea of a *telescope*, i.e., a chain of convergent theory inclusions of the form:

$$(\Omega, B_\Omega) \subseteq (\Sigma_1, E_1 \cup B_1) \subseteq (\Sigma, E \cup B)$$

such that: (i) Ω is the subsignature of operators that were specified as constructors, with the `ctor` atribute, in the functional module specifying $(\Sigma, E \cup B)$, (ii) $B_\Omega \subseteq B$ are the axioms declared for such constructors, (iii) the constructors are true constructors, i.e., for any ground term in the syntax of Σ we have $t!_{E,B} \in T_\Omega$, (iv) any $u \in T_\Omega$ is already in normal form: $u =_{B_\Omega} u!_{E,B}$, and (v) the intermediate theory $(\Sigma_1, E_1 \cup B_1)$ is convergent, has also Ω as its constructors, is FVP, and any A symbol $f \in \Sigma_1$ is also C.

The third and last piece of good news is that, under conditions (i)–(v), satisfiability of QF Σ_1-formulas in $T_{\Sigma/E \cup B}$ is *decidable* [56], which is what we were fishing for; and there is a *theory-generic* satisfiability decision procedure for such formulas, namely, *variant satisfiability* [56]. Of course, at the very least we may have $(\Omega, B_\Omega) = (\Sigma_1, E_1 \cup B_1)$, and in that case just get decidable satisfiability for QF Ω-formulas in $T_{\Sigma/E \cup B}$. But quite often, finding an FVP $(\Sigma_1, E_1 \cup B_1)$ having a strict containment $(\Omega, B_\Omega) \subset (\Sigma_1, E_1 \cup B_1)$ is relatively easy to do. For example, any *selector* functions for the constructors in Ω will automatically be in $(\Sigma_1, E_1 \cup B_1)$ [30].

Eh bien! But how does this theory-generic decision procedure *work*? Recall that solving the problem of the *satisfiability* in the data type $T_{\Sigma/E \cup B}$ of any QF Σ_1-formula φ means to either: (i) effectively exhibiting a *solution*, i.e., a ground substitution ρ such that the ground formula $\varphi\rho$ is *true* in $T_{\Sigma/E \cup B}$ [which by our telescope is the case iff $\varphi\rho$ is *true* in $T_{\Sigma_1/E_1 \cup B_1}$], or (ii) effectively showing that there is no such solution. If this problem is solvable, in one blow, we have also solved the *validity problem* for a QF Σ_1-formula φ in $T_{\Sigma/E \cup B}$. That is, we can either: (i) effectively prove that φ is a *theorem* of $T_{\Sigma/E \cup B}$, or (ii) effectively show a counterexample when it is not: since φ will be a *theorem* of $T_{\Sigma/E \cup B}$ iff $\neg\varphi$ is *unsatisfiable* in

$T_{\Sigma/E\cup B}$. We will solve the satisfiability problem for a QF Σ_1-formula φ in $T_{\Sigma/E\cup B}$ by *reducing* it to that of the satisfiability of QF Ω-formulas in T_{Ω/B_Ω}, which we already know how to decide. Since, without loss of generality, we may assume φ in DNF, that is,

$$\varphi \equiv \bigvee_{1\leqslant i\leqslant n} (\bigwedge_{1\leqslant i.j\leqslant n_i} u_{i.j} = v_{i.j} \wedge \bigwedge_{1\leqslant i.k\leqslant m_i} w_{i.k} \neq w'_{i.k})$$

it is enough to decide the satisfiability of a Σ_1-*conjunction of literals* $\bigwedge_{1\leqslant i\leqslant n} u_i = v_i \wedge \bigwedge_{1\leqslant j\leqslant m} w_j \neq w'_j$. But we already *know* how to decide the satisfiability of the positive part by variant unification. Therefore, the problem reduces to solving the satisfiability of:

$$\bigvee_{\alpha\in Unif_{E_1\cup B_1}(\bigwedge_{1\leqslant i\leqslant n} u_i = v_i)} (\bigwedge_{1\leqslant j\leqslant m} w_j \neq w'_j)\alpha$$

That is, it is enough to decide the satisfiability of a Σ_1-*conjunction of disequalities* $\bigwedge_{1\leqslant j\leqslant m} w_j \neq w'_j$. But, as sketched out in Footnote 4, we can view such a conjunction of disequalities as a *term* in the FVP theory $(\Sigma_1^{=?}, E_1 \cup B_1)$, which has $(\Omega^{=?}, B_\Omega)$ as its subspecification of constructors [i.e., $\Omega^{=?}$ contains *true*, $\_\wedge\_$ and $\_\neq\_$ as added constructors]. But, if we now recall the notion of *constructor variants*, this reduces to the equivalent problem of deciding the satisfiability of the disjunction of conjunctions of Ω-disequalities:

$$\bigvee_{1\leqslant i\leqslant n} (\bigwedge_{1\leqslant j\leqslant m} q^i_j \neq r^i_j)$$

in T_{Ω/B_Ω}, where the $\{\bigwedge_{1\leqslant j\leqslant m} q^i_j \neq r^i_j \mid 1 \leqslant i \leqslant n\}$ are the *constructor variants* of the $\Sigma_1^{=?}$-term: $\bigwedge_{1\leqslant j\leqslant m} w_j \neq w'_j$. So, we have reduced the problem to one of QF satisfiability in T_{Ω/B_Ω} and we are done!

To be *really done*, we just need to know how satisfiability of a conjunction of Ω-disequalities $\bigwedge_{1\leqslant j\leqslant m} q_j \neq r_j$ is decided in T_{Ω/B_Ω}. But this is really easy [56]. First of all, we can *reduce* to the case where each variable $x_i : s_i$ in the conjunction ranges over a sort s_i such that $T_{\Omega/B_\Omega,s_i}$ is an *infinite* set: since if any $x_j : s_j$ ranges over a *finite* set $T_{\Omega/B_\Omega,s_j}$, we can replace our conjunction by a disjunction of conjunctions where $x_j : s_j$ has been instantiated in all possible ways by one of the values in the finite set $T_{\Omega/B_\Omega,s_j}$. Under this infinite-sorts assumption, the conjunction $\bigwedge_{1\leqslant j\leqslant m} q_j \neq r_j$ is satisfiable in T_{Ω/B_Ω} iff $q_j \neq_{B_\Omega} r_j$, $1 \leqslant j \leqslant m$, which is a trivial check in Maude.

Presburger Arithmetic on a Paper Napkin. There are entire book chapters on Presburger arithmetic decision procedures. But to give you a feeling for the general applicability of variant satisfiability, the good news is that by now you *already know everything you need to know* to realize that satisfiability of QF formulas in Presburger arithmetic is *decidable*, and to decide any such QF formula by yourself in Maude. The theory of Presburger arithmetic does indeed fit on a paper napkin, as the functional module:

```
fmod PRESBURGER is protecting TRUTH-VALUE .
sort Nat .
ops 0 1 : -> Nat [ctor] .
op _+_ : Nat Nat -> Nat [ctor assoc comm id: 0] .
op _>_ : Nat Nat -> Bool .
vars N M K : Nat .

eq N + 1 + M > N = true [variant] .
eq N > N + M = false [variant] .
endfm
```

which imports TRUTH-VALUE, with just two constants true, false of sort Bool. Note that in PRESBURGER we have just specified natural number addition as the free commutative monoid generated by 1 with 0 as the identity element. This module is FVP, as one can easily check by computing the three variants of the term N > M for its only defined symbol _>_. Furthermore, all its other operators define a subsignature

Ω of constructor symbols, so that it has a constructor subspecification of the form (Ω, ACU). Therefore, satisfiability of QF Ω-formulas in $T_{\Omega/ACU}$ is *decidable*. And so is also the satisfiability of QF formulas in Presburger arithmetic by our theory-generic variant satisfiability procedure. For example, the transitivity law $N > M = \texttt{true} \wedge M > K = \texttt{true} \Rightarrow N > K = \texttt{true}$ is valid, because its negation $N > M = \texttt{true} \wedge M > K = \texttt{true} \wedge N > K \neq \texttt{true}$ is unsatisfiable, since we get a single solution for the variant unification problem:

```
Maude> filtered variant unify in PRESBURGER : N > M =? true /\ M > K =? true .

Unifier 1
N --> 1 + 1 + %1:Nat + %2:Nat + %3:Nat
M --> 1 + %1:Nat + %2:Nat
K --> %2:Nat

No more unifiers.
```

and when we compute the instantiation $(N > K)\theta$ for this unifier θ and reduce it to its normal form we get:

```
Maude> reduce 1 + 1 + %1:Nat + %2:Nat + %3:Nat > %2:Nat .

result Bool: true
```

making the disequality $true \neq true$ unsatisfiable. q.e.d. Of course, since variant satisfiability is a very general *theory-generic* procedure, there is no fair competition possible with a highly optimized *domain-specific* algorithm for Presburger arithmetic. But this is OK for three reasons: (i) as already mentioned, Maude has interfaces to both the CVC4 and Yices SMT solvers, so optimized implementations of Presburger arithmetic are available that way; (ii) variant satisfiability's sweetspot is not in competing with already existing, optimized *domain-specific* decision procedures, but rather in *complementing* such procedures by making SMT solving *extensible* to an infinite class of *user-definable* algebraic data types; and (iii) nevertheless, a variant satisfiability procedure for Presburger arithmetic is not entirely useless: other colleagues and I have used it in various automated deduction applications, and —as we shall see in a moment— it enjoys the non-negligible advantage of having a *seamless integration* with other variant satisfiability decision procedures.

A Decision Procedure for S-Expressions. This might seem like a bad example to pick in order to show the usefulness of variant satisfiability; but it isn't. After all, domain-specific decision procedures for LISP's S-Expressions go back, at least, to the one by the late Derek Oppen [62]; and similar procedures are a dime a dozen in the SMT solving literature. So, why beating a dead horse? Because it isn't dead. The dirty little secret is that all the procedures of this kind I am aware of are *problematic*. Why so? They are problematic in their *corner cases*, namely, in cases when an S-Expression can be *undefined*. For example, according to the LISP 1.5 Programmer's Manual [45], expressions such as `car[A]` or `cdr[A]` for `A` an atom are *undefined*. The problem is that all the S-Expression decision procedures I am aware of are based on either *unsorted* or *many-sorted* first-order logic. But, as my late friend Joseph Goguen and I showed in [58], the problem of *faithfully* specifying data types involving partial functions such as those for the data *selectors* `car` and `cdr` in LISP, *cannot be solved* in unsorted or many-sorted first-order logic.[5] But, as we showed in [58], it is *solved* by specifying such data types in *order-sorted* equational logic; or in the even more general *membership equational logic* [53] used by Maude's functional modules. The upshot of all this is that the existing decision procedures are forced to cut some corners: the answers you will get in such corner cases are anybody's guess or, if documented, they will depend on some arbitrary choices about how to make such partial functions *total* in the undefined cases.

So, the horse is not really dead yet. And there is something to be gained by revisiting this venerable topic of decision procedures for S-Expressions as a representative instance of the much more general

[5] Unless of course such partial functions are represented as *binary relations*, or the specification itself is *changed* by introducing *coercion functions* in the way Goguen and I showed in [29].

problem of having *faithful* decision procedures for algebraic data types with *constructors and selectors*. Furthermore, it gives me a good opportunity to introduce you, dear reader, to the expressive power of *order-sorted* specifications in Maude, which is actually crucial for many variant satisfiability procedures.

LISP is of course an *untyped* language. However, what might be called LISP's *ontology of S-Expressions*, which is part of the lore and essential to know what you are doing when programming in LISP, is captured by the following structure of subsorts of the main sort `SExp`. Since S-Expressions are *parametric* on the type of *Atoms*, which are basic data values, like numbers, Booleans, identifiers, etc., this can be specified in Maude as a *parameterized module* with the `TRIV` parameter theory, which just has an `Elt` parameter sort/type that can be instantiated to any chosen sort/type of basic values, i.e., of atoms.

```
fmod S-EXP{A :: TRIV} is  protecting TRUTH-VALUE .
sorts List NeList NLExp NLPair SExp .
subsorts NeList < List < SExp .
subsorts A$Elt NLPair < NLExp < SExp .
op nil : -> List [ctor] .
op [_._] : SExp SExp -> SExp [ctor] .
op [_._] : SExp List -> NeList [ctor] .
op [_._] : SExp NLExp -> NLPair [ctor] .
op car_ : NeList -> SExp .    *** left selector
op car_ : NLPair -> SExp .    *** left selector
op cdr_ : NeList -> List .    *** right selector
op cdr_ : NLPair -> NLExp .   *** right selector
ops atom? nelist? list? nlpair? nlexp?  : SExp -> Bool .  *** sort preds

var A : A$Elt . var NeL : NeList .  var L : List .
var NLE : NLExp . var NLP : NLPair . var SE : SExp .

eq car[SE . L] = SE [variant] .      eq cdr[SE . L] = L [variant] .
eq car[SE . NLE] = SE [variant] .    eq cdr[SE . NLE] = NLE [variant] .

eq atom?(A) = true [variant] .       eq nelist?(NeL) = true [variant] .
eq atom?(NLP) = false [variant] .    eq nelist?(nil) = false [variant] .
eq atom?(L) = false [variant] .      eq nelist?(NLE) = false [variant] .
eq list?(L) = true [variant] .       eq nlpair?(NLP) = true [variant] .
eq list?(NLE) = false [variant] .    eq nlpair?(A) = false [variant] .
eq nlexp?(NLE) = true [variant] .    eq nlpair?(L) = false [variant] .
eq nlexp?(L) = false [variant] .
endfm
```

This is the only example in this paper that may not fit on a cocktail paper napkin: we may have to unfold one, or to ask our waiter for a dinner paper napkin. The main ideas about the ontology carved out by the above subsort structure can be summarized by the following remarks about LISP lore: (1) An `SExp` is either an Atom (of the parameter sort `A$Elt`), or `nil`, or a binary tree having either atoms or `nil` in its leaves. (2) A `List` is either `nil`, or a binary tree whose rightmost leaf is `nil`. (3) A `NeList` is a non-nil `List`. (4) A `NLExp` is any *non-list* `SExp`. (5) A `NLPair` is any non-atom `NLExp`. Of course, `car` and `cdr` select the left, resp. right, subtrees of any S-Expression that is a binary tree. They make no sense otherwise. The sort predicates have lower case names for their respective sorts: they are `true` for elements of that sort, and `false` otherwise. Thanks to order-sortedness, some operators are *overloaded*.

This module is FVP. Termination is trivial, since all the equations decrease term size; confluence follows from the absence of order-sorted critical pairs; full definition of functions can be easily checked by the method in [47]; and FVP itself can be easily checked by computing variants for each of the defined functions. For example, `car` and `cdr` have two variants each (for either of their typings), and the `list?` predicate has three variants. As already pointed out, it would have been impossible to faithfully model LISP

S-Expressions in unsorted or many-sorted first-order logic. But there is more behind the module's deceptive simplicity: Even if we had *not* specified the `car` and `cdr` selectors that push this data type outside the pale of many-sorted first-order logic, it would still have been impossible to specify predicates like `list?` or `nlexp?` as FVP functions in an unsorted or many-sorted way. The reason for this impossibility is that in such settings these predicates would have to *recurse down the binary tree* to *check* whether the rightmost element is either `nil` or an atom; and this would have pushed those predicate definitions out of the FVP fold. The moral of this story is that order-sorted first-order logic *silently and kindly absorbs into its syntax* a lot of reasoning that would otherwise require quite complex *first-order reasoning*, in the form of deducing implications between unary predicates modeling the non-existent subsorts.

Since the constructors of `S-EXP` do not satisfy any axioms and no equations apply to constructor terms, we are again under the conditions ensuring decidable satisfiability. That is, we have a variant satisfiability procedure for S-Expressions in a *parametric* way, in the same sense as for similar parametric variant satisfiability procedures for lists, compact lists, multisets, sets, and hereditarily finite sets in [56]. What this means in practice is that if we *instantiate* `S-EXP{A :: TRIV}` by choosing a sort of atoms in any FVP data type that also satisfies the variant satisfiability conditions, then, any such instantiation (after checking termination of the equations in the instantiation) is also FVP and does also have decidable satisfiability for its QF formulas. For example, we can instantiate the parameter sort `Elt` in `TRIV` to the `Nat` sort in `PRESBURGER` by defining in Maude a *view* and then instantiating `S-EXP{A :: TRIV}` with this view as follows:

```
view Nat from TRIV to PRESBURGER is
sort Elt to Nat .
endv

fmod NAT-SEXP is
protecting S-EXP{Nat} .
endfm
```

In this instantiated module —whose termination proof is trivial, since all its equations are term-size decreasing— we can decide the validity of both *parametric* theorems like: $NeL = [(car\,NeL)\,.\,(cdr\,NeL)]$, which hold for any instance of the module and could likewise have been defined directly for `S-EXP{A :: TRIV}`, and that of theorems that only make sense for this instantiation, like the implication:

$$atom?(carNLP) = true \wedge atom?(cdrNLP) = true \Rightarrow (car\,NLP) + (cdr\,NLP) > (car\,NLP) \neq false \vee (cdr\,NLP) = 0$$

Let us prove both of these theorems by showing that their corresponding negations are unsatisfiable. In the first example, the only constructor variant of the disequality $NeL \neq [(car\,NeL)\,.\,(cdr\,NeL)]$ is the clearly unsatisfiable disequality $[SE\,.\,L] \neq [SE\,.\,L]$. q.e.d. In the second example we have to verify that the conjunction

$$atom?(carNLP) = true \wedge atom?(cdrNLP) = true \wedge (car\,NLP) + (cdr\,NLP) > (car\,NLP) = false \wedge (cdr\,NLP) \neq 0$$

is unsatisfiable. But the positive part of this conjunction has the single unifier $\theta = \{NLP \mapsto [N\,.\,0]\}$; and then the canonical form of $(cdr\,NLP)\theta \neq 0$ is the unsatisfiable disequality $0 \neq 0$. q.e.d.

Something interesting about this example is the *seamless integration* of the two variant satisfiability decision procedures: the one for `PRESBURGER` and that for `S-EXP{A :: TRIV}`. This is in contrast to the usual Nelson-Oppen (NO) combination procedure [60] required to reason in a combination of theories. No such NO-combination procedure is needed at all for variant satisfiability: we just form the appropriate *union* of theories (in this case by instantiating the `S-EXP{A :: TRIV}` with the `Nat` view), and that's it!

6 Dessert: Narrowing-Based Symbolic Reachability Analysis

By now we have had a fairly substantial sampling of tapas: we should not push this too hard. Let me end on a light, yet interesting, note by explaining to you what symbolic reachability analysis in Maude is about, and some cool things you can do with it. It will be our dessert: a little *divertimento*. We have remained all the time within Maude's sublanguage of functional modules. But, of course, Maude's most unique capability is its declarative programming of concurrent systems by means of rewrite theories in system modules of the form `mod FOO is` $(\Sigma, E \cup B, R)$ `endm`, where the system's local concurrent transitions are specified by the rules R using the `rl` keyword, as opposed to the `eq` keyword used for equations. Such rules need not be terminating, and can be highly non-deterministic. Maude's `rewrite` command can simulate *one* possible execution sequence for such rules in a fair fashion; but there can be many, many more possible executions. For many reasoning purposes, such as, for example, to check that a cryptographic protocol is secure, one can perform *reachability analysis* in Maude to explore *all* states reachable from a given one using Maude's breadth first `search` command.

However, this may not be powerful enough in some cases: for example, if either the set of reachable states or that of initial states is *infinite*. In such cases one can perform *symbolic reachability analysis* using *narrowing* with Maude's `vu-narrow` command. Thanks to our previous Maude tapas this command is now quite easy to explain. Given a symbolic initial state specified by a term $u(x_1,...,x_n)$ describing a, typically infinite, set of initial state instances, what this command does is to build a *narrowing search graph* rooted at $u(x_1,...,x_n)$. But there are three main differences with equational narrowing: (1) now we narrow symbolic expressions, not with equations E, but *with transition rules* in R; (2) for each narrowing step, instead of performing B-unification as before, we now perform $E \cup B$-unification with all the equations in the rewrite theory; and (3) we check if we have *reached* a goal term $v(y_1,...,y_n)$ using $E \cup B$-unification. There are just two restrictions: (i) to be practical, we want to remain *finitely branching*, so we require the equations $E \cup B$ to be FVP to make sure the number of $E \cup B$-unifiers is finite; and (ii) we also assume that the rules in R are *topmost* —i.e., that they rewrite the entire state—, which is easy to achieve in practice by a theory transformation and ensures completeness of the analysis. The command has the form:

```
vu-narrow [n] in FOO :  u(x1,...,xn) =>* v(y1,...,ym) .
```

where n is the number of desired solutions, $u(x_1,...,x_n)$ is the pattern for initial states, and $v(y_1,...,y_n)$ is the pattern describing the set of states that we wish to reach —or to show that we cannot reach, if they are "bad" states. The meaning of this query is then to seek an answer to the following question:

> *Is there an instance of the set of initial states symbolically specified by* $u(x_1,...,x_n)$ *from which we can reach an instance of the set of target states symbolically specified by* $v(y_1,...,y_n)$ *by a sequence of transitions from R in the* `FOO` *module?* [$u(x_1,...,x_n)$ and $v(y_1,...,y_n)$ can share some variables]

What Maude's `vu-narrow` command provides is a *complete* method to get answers for such a question: if an answer exists, we are guaranteed —except for the usual memory and time limitations— to find it. The most common examples of this method involve analyzing the reachability properties of some concurrent system. For example, the Maude-NPA tool [26] uses this kind of narrowing-based symbolic reachability analysis (with some additional optimizations), to symbolically analyze security properties of cryptographic protocols. But I wish to present a completely different kind of example, namely, a Logic Programming (LP) interpreter, because it shows that rewriting logic and Maude have good properties not only as a *semantic framework* to naturally specify and program concurrent systems, but also as a *logical framework* [43] in which a logic's inference rules can be naturally represented as *rewrite rules*. In this case, the inference system in question is that of *Horn Logic*; and we get for free an LP interpreter whose core is the following `LP` module importing the quoted identifiers module `QID` with sort `Qid`:

```
fmod LP is protecting QID .
 sorts U UList Query .
 subsorts Qid < U < UList .
 op true : -> UList .          *** true as "nil"
 op _,_ : UList UList -> UList [assoc id: true] .
 op _[_] : Qid UList -> U .  *** term constructor
 op {_} : UList -> Query .
endfm
```

This tiny functional module is all we need to define an interpreter for Logic Programming (LP) [without negation as failure]; i.e., for computing with Horn Logic programs. Terms of sort `U` provide a universal language for *atomic predicates*. For example, the binary atomic predicate $s(s(0)) > s(0)$ will be here represented as the term `'>['s['s['0]],'s['0]]`. The sort `Query` is used for users of the LP interpreter to enter queries. Such queries ask for a witness proving an existential formula of the form:

$$(\exists x_1,...,x_n)\ B_1 \wedge ... \wedge B_k$$

which is here represented by a term `{B1,...,Bk}` of sort `Query`. Prolog's depth first search makes it incomplete. But this interpreter will be *complete*, i.e., if an answer to a query exists, it will be found. Let me explain how we execute a Horn Logic program, i.e., a collection of *Horn clauses*, either of the form A, some atomic predicate, or implications of the form: $A_1 \wedge ... \wedge A_n \rightarrow A$, with $A_1,...,A_n,A$ atomic predicates. If we think of *true* as the empty conjunction, we can view all such Horn clauses as implications, since A is equivalent to $true \rightarrow A$. In LP, and also in proof theory, the conjunction symbol is often represented just by a comma: $\_,\_$ and therefore a Horn clause looks either like $true \rightarrow A$ or like $A_1,...,A_n \rightarrow A$. But in logic we often take the goal we want to prove as our starting point and apply the inference rules *in reverse* to search for a proof of the goal. Therefore, to compute with a set of Horn clauses, i.e., with an LP program, we will use the clauses *in reverse* as rewrite rules: $A \rightarrow true$ and $A \rightarrow A_1,...,A_n$. This representation would be just fine for us to get an LP interpreter: we could make $\_,\_$ associative-commutative with identity *true* and perform symbolic reachability analysis from our goal $B_1,...,B_k$ —which we want to existentially prove by finding a witness using the reversed rewrite rules of type $A \rightarrow true$ and $A \rightarrow A_1,...,A_n$— by trying to reach the term *true*, and thus a proof. This would *work* and would be complete; but it would be quite *inefficient*, because the interpreter would waste a lot of time performing *redundant* symbolic searches. We can achieve a much more efficient interpreter by introducing two seemingly small optimizations: (1) Make $\_,\_$ just AU, instead of ACU. This is harmless, since all lefthand sides of the reverse rules are single atoms. So, they can be applied *anywhere*, i.e., the C axiom is *unnecessary*. (2) By using the operator $\{\_\}$ in the above `LP` module, we can further impose a *left to right order* in searching for proofs of each of our atom goals *one at a time*. This will provide great efficiency. This suggests representing a clause in reverse of the form $A \rightarrow true$ as the "clause in context" rewrite rule $\{A,L\} \rightarrow \{L\}$, taking advantage of the AU axioms, with L a variable of sort `UList`. Likewise, we will represent a clause in reverse $A \rightarrow A_1,...,A_n$ as the "clause in context" $\{A,L\} \rightarrow \{A_1,...,A_n,L\}$. This is just what we will do. For example, the following Horn clauses define the reverse [mirror image] of a binary tree and a *palindrome* predicate on binary trees, where $\_\wedge\_$ is the binary tree constructor and with the elements on tree leaves quoted identifiers; so Q ranges over quoted identifiers:

- $rev(Q,Q)$
- $rev(T_1,T_4),rev(T_2,T_3) \rightarrow rev((T_1 \wedge T_2),(T_3 \wedge T_4))$
- $rev(T,T) \rightarrow pal(T)$

Using our "reversed clauses in context" transformation to compute with these clauses in search for a proof of an existential query, we get the rewrite theory in the following Maude system module, where the `[narrowing]` attribute instructs Maude that the so-marked rules will be used in narrowing search:

```
mod TREE-REVERSE&PALINDROME is  protecting LP .
var Q : Qid .  vars T T' T1 T2 T3 T4 : U .  var L : UList .

rl {('rev[Q,Q]),L} => {L}                   [narrowing] .
rl {('rev[('^[T1,T2]),('^[T3,T4])]),L}
    => {('rev[T1,T4]),('rev[T2,T3]),L}     [narrowing] .
rl {('pal[T]),L} => {('rev[T,T]),L}        [narrowing] .
endm
```

Solving queries for this logic program *is just narrowing with the program's rules!* (in this case *modulo* AU). And, thanks to the completeness of narrowing, such query solving is *complete*. For example:

```
Maude> vu-narrow [1] in TREE-REVERSE&PALINDROME :
{’rev[(’^[(’^[’a,’b]),(’^[’c,’d])]),T]} =>* {true} .

Solution 1
state: {true}
accumulated substitution:
T --> ’^[(’^[’d,’c]),(’^[’b,’a])]

Maude> vu-narrow [2] in TREE-REVERSE&PALINDROME :
{’rev[(’^[(’^[’a,’b]),T’]),T]} =>* {true} .

Solution 1
state: {true}
accumulated substitution:
T’ --> @1:Qid
T --> ’^[@1:Qid,(’^[’b,’a])]
variant unifier:

Solution 2
state: {true}
accumulated substitution:
T’ --> ’^[@2:Qid,@1:Qid]
T --> ’^[(’^[@1:Qid,@2:Qid]),(’^[’b,’a])]

Maude> vu-narrow [1] in TREE-REVERSE&PALINDROME :
{’pal[(’^[(’^[’a,’b]),(’^[’c,’d])])]} =>* {true} .

No solution.

Maude> vu-narrow [1] in TREE-REVERSE&PALINDROME :
{’pal[(’^[(’^[’a,’b]),(’^[’b,’a])])]} =>* {true} .

Solution 1
state: {true}
```

7 Further Reading

These tapas have been a way of introducing you, dear reader, in an informal, high-bandwith way to some symbolic aspects of Maude that you might find useful. As agreed, I have tried to kept technical details to a bare minimum: just sufficient for an intelligent conversation with someone having a CS background to be *meaningful*. Now is the time to explain to you how a few gaps we had to skirt can be filled in. I focus on Maude in Section 7.1, and discuss broader mathematical background readings in Section 7.2.

7.1 Further Reading on Maude

The most up-to-date Maude journal paper —also emphasizing symbolic aspects— and covering other aspects such as Maude’s **strategy language** and Maude’s approach to **concurrent object-oriented programming** and various Maude *external objects* —that allow Maude programs to be executed in a distributed manner and interact with external entities— is [20]. The Maude book [14] is dated —since important new features were added later— but is still useful for those parts it covers and its tutorial examples. For teaching formal methods using Maude, Peter Ölvecky’s book [61] is an excellent textbook emphasizing distributed system applications. In particular, [20], [14] and [61] provide more precise definitions of **rewriting modulo** *B* and a wealth of examples of both functional and system modules, including *parameterized* ones such as the `S-EXP{A :: TRIV}` one we already encountered, and the use of the `reduce` and `rewrite` commands. For *executability conditions* and how to check them, for both functional and system modules, see [24, 22, 32]. References [14] and [61] also provide good explanations and examples to understand the use of Maude's breadth first `search` command, and how `search` supports a basic, yet very useful, form of **model checking** verification. They also explain and illustrate well the more sophisticated **LTL temporal logic model checking** also supported directly by Maude.

Something important that did no come up in our conversation over tapas is **reflection**. It did come up subliminally in *theory transformations* like $(\Sigma, E \cup B) \mapsto (\Sigma^{=?}, E^{=?} \cup B)$, or in transforming a Horn theory into a Maude system module. The point about reflection is that any such transformations can be performed *inside Maude*, because Maude's `META-LEVEL` module supports *meta-programming*, i.e., writing programs that manipulate other programs. This is not some kind of useful hack, but a piece of mathematics: the efficient exploitation inside Maude of the fact that both rewriting logic and its underlying equational logic are *reflective* [16], i.e., have *universal theories* that can faithfully represent any theories [including themselves] as *data*, as well as faithfully simulating deduction in them. The reason why this may be of interest to you is because —combined with the symbolic features I have explained— reflection makes it very easy to build many *formal tools*, not just for Maude itself, but for many other logics. Of course, in the Maude team we aggressively practice *dogfooding*, so all the Maude formal verification tools have been built this way; but other researchers use Maude in the same way for many other logics and languages. The Maude book [14], and [20], are good sources to learn more about reflection in Maude.

To learn more about how to use **unification, variants, and narrowing-based reachability analysis** in Maude, the best sources at present are the journal paper [20], the conference paper [21], and the Maude 3.1 Manual [15]. I discuss theoretical foundations for these and other topics in Section 7.2.

There are many other aspects of Maude and rewriting logic, and many other applications that I could not discuss here. A somewhat dated but still useful survey of rewriting logic, including also references to many applications developed in Maude, is the 2012 paper [54].

7.2 Further Background Reading

I focus here on answering the question: *Where can I learn more about the mathematical foundations of the topics we have discussed over tapas?* This is different from questions about Maude itself, which, hopefully, were answered in Section 7.1.

Logics. The three main logics involved are: (i) *equational logic*; (ii) its extension to *first-order logic*; and (iii) *rewriting logic*. Both (ii) and (iii) are parametric on the equational logic chosen. Since Maude functional modules specify algebraic data types, the million-dollar question is: *What is a good logic to specify algebraic data types?* This question is highly non-trivial, due to the presence of partial functions in many data types. Joseph Goguen and I proposed *order-sorted equational logic* in [29], further developed in [53]. I later proposed the extension of order-sorted equational logic to *membership equational logic* in [53], and developed its computational logic aspects and its rewriting techniques jointly with Adel Bouhoula and Jean-Pierre Jouannaud in [9]. Maude's functional modules are based on membership equational logic; but many examples can be specified as order-sorted theories. Any equational logic is just a fragment of a corresponding first-order logic. For order-sorted logic this is explained in detail in, e.g., [69]. For simplicity of exposition, *rewriting logic* was first presented in [52] as having unsorted equational logic as its sublogic. But from the beginning the intention was to base it on order-sorted equational logic; and it was further extended, based on membership equational logic, in [10]. A latest extension allowing quantifier-free formulas in the conditions of conditional rules is presented in [57].

Rewriting Modulo *B*, and Rewriting in Rewrite Theories. I have not touched upon *conditional* rewriting, which generalizes the unconditional case and is supported by Maude. For the semantics of conditional rewriting modulo *B* in *convergent* order-sorted equational theories, a quite comprehensive reference is [41].

I have cheated a little by saying that convergent means Church-Rosser and terminating: in the modulo B case the additional requirement of *B-coherence* [37, 55] is needed; but this is automatically enforced by the Maude implementation. Furthermore, in the order-sorted case *sort-decreasingness* (see, e.g., [41]), i.e., that the sorts of terms remain the same or go down by rewriting, is also needed for convergence. The key theorem for equational rewriting is that if $(\Sigma, E \cup B)$ is convergent, then we have the *Church-Rosser Equivalence*:

$$u =_{E \cup B} v \;\Leftrightarrow\; u!_{E,B} =_B v!_{E,B}$$

A very general formulation of this equivalence for the conditional order-sorted case can be found in [41]. As already mentioned, rewriting in conditional theories in membership equational logics has been studied in [9].

For a rewrite theory, $\mathcal{R} = (\Sigma, E \cup B, R)$, rewriting with transition rules R should happen *modulo $E \cup B$*. But this is of course very hard to implement, since $E \cup B$-equality may even be undecidable. Furthermore, both the equations E and the rules R can be *conditional*. However, under the natural assumption that $(\Sigma, E \cup B)$ is convergent, a simple requirement called *coherence* of R with E modulo B [73, 24] ensures that the unmanageable relation $\rightarrow_{R/(E \cup B)}$ can be faithfully simulated by the much simpler relations $\rightarrow_{R,B}$ and $\rightarrow_{E,B}$. This is what the Maude implementation supports, requiring system modules to be coherent.

Unification, Narrowing, Variants, and Variant Unification. *Unification* is technical jargon for solving equations in an algebra. For algebras whose elements are numbers, this goes back to Classical Greece, where many of these problems arose in conjunction with geometrical constructions, e.g., measuring the diagonal of a unit square. It was advanced by the Arabs, who coined the word "Algebra" for this business, and further developed by the Italians, Newton, Galois, Gauss, the Emmy Noether school, and so on. Two fundamental problems about solving equations in numerical domains were settled in the 20th Century: (i) the effective solvability of polynomial equations and inequalities in any real-closed field, and in particular in the reals, thanks to the Tarski-Seidenberg Theorem [72, 67] —which actually decides the satisfiability of any first-order formula in this language—, and (ii) the inexistence of a general algorithm to solve polynomial equations in the integers —the so-called *diophantine equations*, after Diophantus—, thanks to Matiyasevich's negative answer to Hilbert's 10th Problem [44]. But with the rise of symbolic logic in the 20th Century, the need naturally arose to *solve equations in term algebras*, i.e., in T_Σ or $T_\Sigma(X)$ for variables X: it amounts to the same if Σ has constants. This problem was solved by Jacques Herbrand in his thesis (see [33], pg. 148). In Computer Science, Herbrand's algorithm was rediscovered independently by Alan Robinson, who called it "unification," as the main workhorse for *resolution*: his breakthrough in automated theorem proving [65]. Since resolution was based on first-order logic *without* equality, the issue of how to "build in" equational theories in resolution provers so as to avoid falling into the Turing tarpits was recognized as a pressing one by Gordon Plotkin [64], who proceeded to give an A-unification algorithm for this purpose in [64]. Independently, Makanin in Russia provided a different A-unification algorithm in [42]. Likewise, Peterson and Stickel gave an AC-unification algorithm in [63]. This raised the general E-unification problem, that is, how to solve equations in the data type $T_{\Sigma/E}$, or equivalently in $T_{\Sigma/E}(X)$, for various E: see [36, 6, 5] for three surveys. The treatment of E-unification was unsorted. But this is too restrictive for the reasons already mentioned above. Therefore, the need for more general *order-sorted E-unification* algorithms arose naturally and was answered in [66, 59, 71]. Additional advances were made in [31] and —crucially for the efficiency of Maude's implementation of order-sorted B-unification— in [25].

Narrowing also emerged from efforts to make resolution theorem provers reason efficiently about equality. Specifically, it was introduced by Slagle [70] as an efficient kind of *paramodulation*, and was further elaborated by Lankford as a component of a resolution-with-equality strategy assuming convergent equations [39]. Hullot further advanced the narrowing ideas, proposed his *basic narrowing* strategy, and explored under some restrictions the notion of narrowing modulo axioms B for a convergent theory $(\Sigma, E \cup B)$ in [34]. A more systematic generalization to this case was carried out by J.-P. Jouannaud, C. Kirchner and H. Kirchner in [35], assuming a B-unification algorithm. The generalization to narrowing with convergent order-sorted *conditional* equational theories modulo B has been carried out in [11].

Both Fay [28] and Hullot [34] realized that narrowing could be used to compute E-unifiers of the convergent equations E used as rules in the narrowing. Furthermore, Hullot discussed in [34] how $E \cup B$-unification algorithms could be obtained via narrowing modulo B for $(\Sigma, E \cup B)$ convergent in some cases. Again, a more systematic extension of narrowing-based $E \cup B$-unification was carried out by J.-P. Jouannaud, C. Kirchner and H. Kirchner in [35], and was later extended to $E \cup B$-unification for convergent order-sorted *conditional* equational theories in [11]. However, narrowing-based $E \cup B$-unification suffers from two main drawbacks: (i) since the conditions for termination of narrowing are very restrictive, what narrowing-based $E \cup B$-unification generally provides is only a *semi-algorithm*: if a $E \cup B$-unifier exists, it will be found in a finite number of steps —up to pragmatic time and space limitations; but if it does *not* exist, we may never find out, making $E \cup B$-unifiability undecidable in general by this method; and (ii) since some axioms B can give rise to huge numbers of B-unifiers, these algorithms can suffer serious combinatorial explosions. Here is where variants, discussed next, can make a big difference.

Comon and Delaune proposed the notion of *variant* and studied its properties in [18]. *Folding variant narrowing* and *variant unification* were defined and developed in [27]. Several alternative notions of variant, their relationships, and ways of checking FVP are discussed in [12]. The extension of the properties and methods of variants modulo axioms B when B-unification can have an infinite set of B-unifiers has been initiated in [49]. As already explained in Section 4, $E \cup B$-unification with the folding variant narrowing strategy has two key advantages: (i) it terminates with a complete finite set of $E \cup B$-unifiers iff $(\Sigma, R \cup B)$ is FVP, and (ii) its search space and its efficiency are much better than standard narrowing-based $E \cup B$-unification. There are many applications of variants and variant unification to, e.g., cryptographic protocol analysis, e.g., [18, 13, 26, 46], program termination [23], SMT solving, e.g., [56, 68], partial evaluation, e.g., [3], program transformation and symbolic model checking, e.g., [57, 7], and theorem proving, e.g., [69, 50].

Variant Satisfiability. The foundations and many examples can be found in [56]. Decidable QF satisfiability in $T_{\Sigma/B}$ whenever any A symbol $f \in \Sigma$ is also C, generalizes that of $T_{\Sigma/AC}$ in [17]. Variant satisfiability algorithms are studied in [68]. An extension to specifications with predicates, plus variant satisfiability of data types with constructors and selectors can be found in [30]. For variant satisfiability examples with $B = A$ see [48]. For theorem proving applications see [69, 50].

Narrowing-Based Reachability Analysis. Narrowing was developed as an automated deduction method for *equational reasoning*. The idea that narrowing based $E \cup B$-unification could be used to perform *symbolic reachability analysis* in a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ by narrowing symbolic states with *transition rules R modulo $E \cup B$* was proposed in [51], with cryptographic protocol analysis as an application in mind. In fact, the most impressive application of this technique is the Maude-NPA tool for analysis of cryptographic protocols (see [26] for a tutorial, and more recent references in DBLP). The extension of this technique from reachability analysis to symbolic LTL model checking —with a Maude-based tool supporting it— can be found in [7]. Symbolic reachability analysis with very general conditional rules is studied in [57].

Acknowledgements. I thank the BOPL organizers for giving me the opportunity of presenting these ideas as a BOPL joint invited speaker. I chose the talk's topic having in mind the interests of the various BOPL participants and, in spite of the pandemic, found the online discussions very helpful and stimulating. The ideas I have presented are based on joint work with various colleagues. The symbolic aspects of Maude are part of a long and extremely active effort by the members of the Maude Team; they owe much to Steven Eker's high-performance implementation of its features. Folding variant narrowing is joint work with Santiago Escobar and Ralf Sasse. Variant-based satisfiability has been advanced in joint work with Stephen Skeirik and Raúl Gutiérrez. The Maude-NPA has been developed in joint work with Catherine Meadows, Santiago Escobar, and Ph.D. students at Illinois, Valencia, and Oslo. Maude's Symbolic LTL Model Checker is joint work with Kyungmin Bae and Santiago Escobar. Last but not least, the work on generalization, homeomorphic embedding and variant-based partial evaluation of Maude programs is joint research with María Alpuente, Angel Cuenca-Ortega, Santiago Escobar and Julia Sapiña at TU Valencia, and Demis Ballis at the University of Udine. Given the long list, I hope I have not missed anybody, and apologize in advance if that were inadvertently the case. I warmly thank María Alpuente, Francisco Durán, Santiago Escobar, Maribel Fernádez, Salvador Lucas, Narciso Martí-Oliet, Rubén Rubio and Carolyn Talcott for their very helpful suggestions to improve the manuscript. The research reported herein has been partially supported by NRL under contract N00173-17-1-G002.

References

1. Alpuente, M., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Order-sorted homeomorphic embedding modulo combinations of associativity and/or commutativity axioms. Fundamenta Informaticae **177**, 297–329 (2020)
2. Alpuente, M., Ballis, D., Cuenca-Ortega, A., Escobar, S., Meseguer, J.: Acuos^2: A high-performance system for modular ACU generalization with subtyping and inheritance. In: Proc. Logics in Artificial Intelligence, JELIA 2019. Lecture Notes in Computer Science, vol. 11468, pp. 171–181. Springer (2019)
3. 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)
4. Alpuente, M., Escobar, S., Espert, J., Meseguer, J.: A modular order-sorted equational generalization algorithm. Inf. Comput. **235**, 98–136 (2014)
5. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning. Elsevier (1999)
6. Baader, F., Siekmann, J.H.: Unification theory. In: Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2, pp. 41–126. Oxford University Press (1994)
7. Bae, K., Escobar, S., Meseguer, J.: Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In: Rewriting Techniques and Applications (RTA'13). LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
8. Bouchard, C., Gero, K.A., Lynch, C., Narendran, P.: On forward closure and the finite variant property. In: Proc. FroCoS 2013. LNCS, vol. 8152, pp. 327–342. Springer (2013)
9. Bouhoula, A., Jouannaud, J.P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science **236**, 35–132 (2000)
10. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1-3), 386–414 (2006)
11. Cholewa, A., Escobar, S., Meseguer, J.: Constrained narrowing for conditional equational theories modulo axioms. Science of Computer Programming **112**, 24–57 (2015)
12. Cholewa, A., Meseguer, J., Escobar, S.: Variants of variants and the finite variant property. Tech. rep., CS Dept. University of Illinois at Urbana-Champaign (February 2014), available at `http://hdl.handle.net/2142/47117`
13. Ciobaca., S.: Verification of Composition of Security Protocols with Applications to Electronic Voting. Ph.D. thesis, ENS Cachan (2011)
14. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework. Springer LNCS Vol. 4350 (2007)
15. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude Manual (Version 3.1), october 2020, http://maude.cs.uiuc.edu
16. Clavel, M., Meseguer, J., Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. Theoretical Computer Science **373**, 70–91 (2007)
17. Comon, H.: Unification et disunification: Théorie et applications. Ph.D. thesis, Institute National Polytechnique de Grenoble, France (1988)
18. Comon-Lundth, H., Delaune, S.: The finite variant property: how to get rid of some algebraic properties, in Proc *RTA'05*, Springer LNCS 3467, 294–307, 2005
19. CVC4: Availabla at https://cvc4.github.io
20. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebr. Meth. Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497, https://doi.org/10.1016/j.jlamp.2019.100497
21. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Associative unification and symbolic reasoning modulo associativity in maude. In: 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. 98–114. Springer (2018)
22. Durán, F., Lucas, S., Meseguer, J.: MTT: The Maude Termination Tool (system description). In: IJCAR 2008. Lecture Notes in Computer Science, vol. 5195, pp. 313–319. Springer (2008)

23. Durán, F., Lucas, S., Meseguer, J.: Termination modulo combinations of equational theories. In: Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5749, pp. 246–262. Springer (2009)
24. Durán, F., Meseguer, J.: On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. J. Algebraic and Logic Programming **81**, 816–850 (2012)
25. Eker, S.: Fast sort computations for order-sorted matching and unification. In: Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday. vol. 7000, pp. 299–314. Springer LNCS (2011)
26. 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, LNCS, vol. 5705, pp. 1–50. Springer (2009)
27. Escobar, S., Sasse, R., Meseguer, J.: Folding variant narrowing and optimal variant termination. J. Algebraic and Logic Programming **81**, 898–928 (2012)
28. Fay, M.: First-order unification in an equational theory. In: Proc. Fourth Workshop on Automated Deduction, Austin, Texas, pp. 161–167 (1979)
29. Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science **105**, 217–273 (1992)
30. Gutiérrez, R., Meseguer, J.: Variant-based decidable satisfiability in initial algebras with predicates. In: Proc. Logic-Based Program Synthesis and Transformation – LOPSTR 2017. Lecture Notes in Computer Science, vol. 10855, pp. 306–322. Springer (2017)
31. Hendrix, J., Meseguer, J.: Order-sorted equational unification revisited. Electr. Notes Theor. Comput. Sci. **290**, 37–50 (2012)
32. Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Automated Reasoning, Third International Joint Conference, IJCAR 2006. pp. 151–155 (2006)
33. Herbrand, J.: Logical Writings. Reidel (1971)
34. Hullot, J.M.: Canonical forms and unification. In: Proc. Fifth Conference on Automated Deduction, LNCS, vol. 87, pp. 318–334. Springer (1980)
35. Jouannaud, J.P., Kirchner, C., Kirchner, H.: Incremental construction of unification algorithms in equational theories. In: Proc. ICALP'83. pp. 361–373. Springer LNCS 154 (1983)
36. Jouannaud, J.P., Kirchner, C.: Solving equations in abstract algebras: A rule-based survey of unification. In: Computational Logic - Essays in Honor of Alan Robinson. pp. 257–321. MIT Press (1991)
37. Jouannaud, J.P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM Journal of Computing **15**, 1155–1194 (November 1986)
38. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
39. Lankford, D.S.: Canonical inference. Tech. Rep. ATP-32, Southwestn Univ. (1975)
40. Levi, G., Sirovich, F.: Proving program properties, symbolic evaluation and logical procedural semantics. In: Proc. Mathematical Foundations of Computer Science 1975, 4th Symposium. Lecture Notes in Computer Science, vol. 32, pp. 294–301. Springer (1975)
41. Lucas, S., Meseguer, J.: Normal forms and normal theories in conditional rewriting. J. Log. Algebr. Meth. Program. **85**(1), 67–97 (2016)
42. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math USSR Sbornik **32**(2), 129–198 (1977)
43. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. In: Gabbay, D., Guenthner, F. (eds.) Handbook of Philosophical Logic, 2nd. Edition, pp. 1–87. Kluwer Academic Publishers (2002), first published as SRI Tech. Report SRI-CSL-93-05, August 1993
44. Matiyasevich, Y.V.: Hilbert's 10th Problem. MIT Press (1993)
45. McCarthy, J., Abrahams, P., Edwards, D., Hart, T., Levin, M.: LISP 1.5 Programmer's Manual. MIT Press (1985)
46. Meier, S., Schmidt, B., Cremers, C., Basin, D.A.: The TAMARIN prover for the symbolic analysis of security protocols. In: Proc. CAV 2013. Lecture Notes in Computer Science, vol. 8044, pp. 696–701. Springer (2013)

47. Meseguer, J.: Order-sorted parameterization and induction. In: Semantics and Algebraic Specification. Lecture Notes in Computer Science, vol. 5700, pp. 43–80. Springer (2009)
48. Meseguer, J.: Variant satisfiability of parameterized strings. In: Proc. WRLA 2020. LNCS, vol. 12328, pp. 96–113. Springer (2020)
49. Meseguer, J.: Variants in the infinitary unification wonderland. In: Proc. WRLA 2020. LNCS, vol. 12328, pp. 75–95. Springer (2020)
50. Meseguer, J., Skeirik, S.: Inductive reasoning with equality predicates, contextual rewriting and variant-based simplification. In: Proc. WRLA 2020. LNCS, vol. 12328, pp. 114–135. Springer (2020)
51. Meseguer, J., Thati, P.: Symbolic reachability analysis using narrowing and its application to the verification of cryptographic protocols. J. Higher-Order and Symbolic Computation **20**(1–2), 123–160 (2007)
52. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
53. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. pp. 18–61. Springer LNCS 1376 (1998)
54. Meseguer, J.: Twenty years of rewriting logic. J. Algebraic and Logic Programming **81**, 721–781 (2012)
55. Meseguer, J.: Strict coherence of conditional rewriting modulo axioms. Theor. Comput. Sci. **672**, 1–35 (2017)
56. Meseguer, J.: Variant-based satisfiability in initial algebras. Sci. Comput. Program. **154**, 3–41 (2018)
57. Meseguer, J.: Generalized rewrite theories, coherence completion and symbolic methods. Journal of Logic and Algebraic Methods in Programming (2019), in this issue
58. Meseguer, J., Goguen, J.: Order-sorted algebra solves the constructor-selector, multiple representation and coercion problems. Information and Computation **103**(1), 114–158 (1993)
59. Meseguer, J., Goguen, J., Smolka, G.: Order-sorted unification. J. Symbolic Computation **8**, 383–413 (1989)
60. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
61. Ölveczky, P.C.: Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude. Undergraduate Topics in Computer Science, Springer (2017)
62. Oppen, D.C.: Complexity, convexity and combinations of theories. Theor. Comput. Sci. **12**, 291–302 (1980)
63. Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. Journal of the Association Computing Machinery **28**(2), 233–264 (1981)
64. Plotkin, G.: Building-in equational theories. Machine Intelligence **7**, 73–90 (November 1972)
65. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery **12**, 23–41 (1965)
66. Schmidt-Schauß, M.: Computational Aspects of an Order-Sorted Logic with Term Declarations, Lecture Notes in Computer Science, vol. 395. Springer (1989)
67. Seidenberg, A.: A new decision method for elementary algebra. Annals of Mathematics **60**, 365–374 (1954)
68. Skeirik, S., Meseguer, J.: Metalevel algorithms for variant satisfiability. J. Log. Algebr. Meth. Program. **96**, 81–110 (2018)
69. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. Fundam. Inform. **173**(4), 315–382 (2020)
70. Slagle, J.R.: Automated theorem-proving for theories with simplifiers commutativity, and associativity. J. ACM **21**(4), 622–642 (1974)
71. Smolka, G., Nutt, W., Goguen, J., Meseguer, J.: Order-sorted equational computation. In: Nivat, M., Aït-Kaci, H. (eds.) Resolution of Equations in Algebraic Structures, vol. 2, pp. 297–367. Academic Press (1989)
72. Tarski, A.: A Decision Method for Elementary Algebra and Geometry. University of California Press (1951), prepared with the assistance of J.C.C. McKinsey.
73. Viry, P.: Equational rules for rewriting logic. Theoretical Computer Science **285**, 487–517 (2002)
74. Yices: Availabla at https://yices.csl.sri.com