Inspecting Maude variants with GLINTS*

M. ALPUENTE, S. ESCOBAR and J. SAPIÑA

DSIC-ELP, Universitat Politècnica de València, Spain
Universidad de Guayaquil, Guayaquil, Ecuador.
(*e-mail:* `alpuente@dsic.upv.es,sescobar@dsic.upv.es,jsapina@dsic.upv.es`)

A. CUENCA-ORTEGA

DSIC-ELP, Universitat Politècnica de València, Spain
Universidad de Guayaquil, Ecuador
(*e-mail:* `angel.cuencao@ug.edu.ec`)

Abstract

This paper introduces GLINTS, a graphical tool for exploring variant narrowing computations in Maude. The most recent version of Maude, version 2.7.1, provides quite sophisticated unification features, including order-sorted equational unification for convergent theories modulo axioms such as associativity, commutativity, and identity. This novel equational unification relies on built-in generation of the set of *variants* of a term t, i.e., the canonical form of $t\sigma$ for a computed substitution σ. Variant generation relies on a novel narrowing strategy called *folding variant narrowing* that opens up new applications in formal reasoning, theorem proving, testing, protocol analysis, and model checking, especially when the theory satisfies the *finite variant property*, i.e., there is a finite number of most general variants for every term in the theory. However, variant narrowing computations can be extremely involved and are simply presented in text format by Maude, often being too heavy to be debugged or even understood. The GLINTS system provides support for (i) determining whether a given theory satisfies the finite variant property, (ii) thoroughly exploring variant narrowing computations, (iii) automatic checking of node *embedding* and *closedness* modulo axioms, and (iv) querying and inspecting selected parts of the variant trees.

KEYWORDS: Rewriting Logic, Narrowing, Variant, Maude, Embedding, Finite Variant Property

1 Introduction

Narrowing is a symbolic execution mechanism that generalizes term rewriting by allowing free variables in terms (as in logic programming) and handles them by using unification (instead of pattern matching) to non-deterministically reduce these terms. Originally introduced in the context of theorem proving, narrowing is complete in the

sense of logic programming (computation of answers) and functional programming (computation of irreducible forms) so that efficient versions of narrowing have been adopted as the operational principle of so-called multi-paradigm (functional logic) programming languages (see, e.g., Hanus (2013)). In the last few years, there has been a resurgence of narrowing in many application areas such as equational unification, state-space exploration, protocol analysis, termination analysis, theorem proving, deductive verification, model transformation, testing, constraint solving, and model checking. To a large extent, the growing interest in narrowing is motivated by the recent takeoff of symbolic execution applications and the availability of efficient narrowing implementations.

Maude is a language and a system that efficiently implements Rewriting Logic (Meseguer 1992), which is a logic of change that seamlessly unifies a wide variety of models of concurrency. Thanks to its logical basis, Maude provides a precise mathematical model, which allows it to be used as a declarative language and as a formal verification system. The most recent version of Maude, version 2.7.1 (Clavel *et al.* 2016), provides quite sophisticated narrowing-based features, including order-sorted equational unification for convergent theories modulo a set of commonly occurring axioms such as associativity, commutativity, and identity. This novel equational unification relies on built-in generation of the set of variants of a term t (Durán *et al.* 2016). A *variant* (Comon-Lundh and Delaune 2005) of a term t in the theory $\mathscr{E}$ is the canonical (i.e., irreducible in $\mathscr{E}$) form of $t\sigma$ for a given substitution σ; in symbols, $(\sigma, t\sigma\downarrow_{\mathscr{E}})$. Variants are computed in Maude by using the *folding variant narrowing strategy* (Escobar *et al.* 2012), which adopts from *tabled logic programming* (Chen and Warren 1996) the idea of memoizing calls encountered in a query evaluation (along with their answers) in a set of tables so that, if the call is re-encountered, the information from the table is reused instead of running the call again. This is useful in two ways: it prevents looping, which may ensure termination under suitable conditions, and it filters out redundant derivations to a reachable expression leading to better performance. When a convergent theory satisfies the *finite variant property* (FVP) (i.e., there is a finite number of most general variants for every term in the theory), folding variant narrowing computes a minimal and complete set of most general variants in a finite amount of time. Many theories of interest have the FVP, including theories that give algebraic axiomatizations of cryptographic functions used in communication protocols, where FVP is omni-present.

Maude's variant generation mechanism was originally designed as an aid for order-sorted equational unification modulo axioms and related problems. It delivers the set of most general variants of the given theory, but it does not allow the user to control the process in any way nor does it provide the user with thorough information about the variant computation process. Unfortunately, variant computations delivered by Maude using the folding variant narrowing strategy can be extremely involved and are simply presented in text format, often being too heavy to be debugged or even understood.

Recently, the definition and inspection of equational theories for which the variants are generated has become an interesting application on its own, which requires enhanced support for exploring the variant narrowing computations. For example, Yang *et al.* (2011) considers 20 equational theories for protocol analysis

in the protocol analyzer Maude-NPA. These equational theories represent under- and over-approximations of the theory of homomorphic encryption with different variant generation behaviors (see Yang *et al.* (2011) for details). As another example, Meseguer (2015) considers distinct axiomatizations of several equational theories of interest for boolean satisfiability. Given the huge intricacy of variant computations, in both cases, the development of all these equational theories was painful when considering the time and effort required to analyze the different variant-based properties for the considered versions of the theories. Often, even an ordinary developer who uses (variant) narrowing as a functional logic program execution mechanism needs deeper support than currently provided by Maude.

This paper describes an inspecting tool for variant computations in Maude called GLINTS (*GraphicaL Interactive Narrowing Tree Searcher*) and its implementation. GLINTS does not only visualize the variants generated by Maude; it goes beyond that by showing internal narrowing computations in full detail, including partially computed substitutions, *Ax*-matching and equational normalization steps that are concealed within Maude's variant narrowing and equational rewriting algorithms. Exploration and visualization in GLINTS can be either automatic or interactive, which allows following promising paths in the narrowing tree without exploring irrelevant parts of it. This supports the design of efficient heuristics for some applications. Also, the displayed view can be abstracted when its size requires it, to avoid cluttering the display with unneeded details. Important insights regarding the programs/theories can be gained from controlling the narrowing space exploration. Does the theory have a finite number of variants? How many variants are there? What do these variants look like and how do they compare to each other modulo axioms? (For instance, is one of the nodes *embedded* or structurally subsumed by one of its ancestors? Is the node *closed* or an equational instance of the tree root or input expression?) What is the meta-level representation of a narrowing computation trace? Moreover, it can also help uncover correctness bugs or even unexpected low performance (by showing which patterns have been executed more often or dominate the execution), which might otherwise be very difficult to identify. As far as we know, this is the first graphical tool in the literature for visual inspection of variant narrowing computations modulo axioms.

After introducing the basic ideas of Maude's narrowing-based variant generation in Section 2, we introduce a leading example for describing GLINTS equational reasoning capabilities based on variant narrowing in Section 3. We explain the core functionality of GLINTS and extra inspection features in Section 4. We provide a description of the tool implementation together with some experiments that assess its performance in Section 5. Finally, some related work and further applications are briefly discussed in Section 6.

GLINTS is publicly available at `http://safe-tools.dsic.upv.es/glints`.

2 Narrowing-based variant generation in Maude

Unification is a deductive mechanism that is used in many automated deduction tools and is essential for programming languages with logical capabilities. Although Maude inherited many features from predecessor languages, like OBJ and Eqlog

(Meseguer 2006), for the sake of high performance, the narrowing-based, logic programming capabilities of the equational logic language Eqlog were left behind since the first public release of Maude in 1999. Order-sorted unification and narrowing modulo axioms first became available in 2009 as a part of Maude 2.4, while variant generation, variant-based unification, and folding variant narrowing have only been implemented 20 years later as built-in, highly efficient features in Maude 2.7.1.

Example 1

Let us illustrate the notion of narrowing in Maude by considering the following simple Maude[1] functional module (with no axioms *Ax*) for addition NAT-VARIANT.

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars X Y Z W : Nat .
  eq [1] : 0 + Y = Y [variant] .
  eq [2] : s(X) + Y = s(X + Y) [variant] .
endfm
```

The *reducible expression*, or simply *redex*, s(0)+0 can be simplified into the result s(0) in two rewriting steps as follows: s(0)+0 →[2] s(0+0) →[1] s(0). Similarly, the non-ground term s(0)+W can be simplified into the result s(W) in two rewriting steps as follows: s(0)+W →[2] s(0+W) →[1] s(W). However, even though the term Z+0 cannot be rewritten as it does not match the left-hand side of any equation, it can be *narrowed* into s(0), with *computed answer substitution* $\theta = \{Z \mapsto s(0)\}$, in two narrowing steps as follows: Z+0 $\overset{\sigma_0}{\leadsto}_{[2]}$ s(X+0) $\overset{\sigma_1}{\leadsto}_{[1]}$ s(0), where $\sigma_0 = \{Z \mapsto s(X), Y \mapsto 0\}$ is the most general unifier of Z+0 and the left-hand side s(X)+Y of the applied equation [2], and similarly $\sigma_1 = \{X \mapsto 0, Y' \mapsto 0\}$ is the most general unifier of X+0 and the (renamed-apart) left-hand side 0+Y' of equation [1], and $\theta = (\sigma_0 \sigma_1)_{\restriction \{Z\}}$.

For an equational theory $\mathscr{E} = (\Sigma, E \cup Ax)$ to be executable, its equations *E* must be *convergent* (i.e., confluent, terminating, and coherent modulo the given axioms *Ax*) (Durán and Meseguer 2012). This ensures (1) that every input expression *t* has one (and only one) *canonical* form $t\downarrow_{\mathscr{E}}$; and (2) that the Maude interpreter can implement conditional rewriting $\to_{E \cup Ax}$ as a much simpler relation $\to_{E,Ax}$ (rewriting with *E* modulo *Ax*) that uses the equations of *E* (oriented from left to right) as the only simplification rules, while the equations in *Ax* are just encapsulated within a powerful algorithm of pattern matching modulo *Ax* that is used at each rewrite step with *E*.

---

[1] Maude syntax is almost self-explanatory, using explicit keywords such as fmod, sort, and op to, respectively, introduce a module, a sort (or type), and an operator, together with its *domain → range* typing declaration that appears after the ":" symbol (e.g., op s : Nat -> Nat). The sort of a variable can be given explicitly in any expression or within the variable declaration section. In addition, from Maude 2.7 and later, only equations with the attribute variant are used by the folding variant narrowing strategy, while all the others are only used for rewriting.

Given $\mathscr{E} = (\Sigma, E \cup Ax)$, the (E, Ax)-*variants* of a term t are the set of all pairs $(\sigma, t\sigma\!\downarrow_{\mathscr{E}})$, each one of which consists of a substitution σ and the (E, Ax)-canonical form of $t\sigma$ (Comon-Lundh and Delaune 2005; Escobar *et al.* 2012). Intuitively, the variants of t are the "irreducible patterns" to which t can be symbolically evaluated by applying the (implicitly oriented) equations of E modulo Ax. For instance, there is an infinite number of variants for the term `(0 + Y:Nat)` in the theory `NAT-VARIANT`, e.g., $(id, \texttt{Y:Nat})$, $(\{\texttt{Y:Nat} \mapsto \texttt{0}\}, \texttt{0})$, $(\{\texttt{Y:Nat} \mapsto \texttt{s(Z:Nat)}\}, \texttt{s(Z:Nat)})$, $(\{\texttt{Y:Nat} \mapsto \texttt{s(0)}\}, \texttt{s(0)})$, ...

A preorder relation of generalization between variants provides a notion of *most general variant* and also a notion of completeness of a set of variants. For the term `0 + Y:Nat`, the most general variant is $(id, \texttt{Y:Nat})$ since any other variant can be obtained by equational instantiation.

Example 2

Consider the following theory that declares the two Boolean constants `true` and `false`. The key thing to note are the special attributes `assoc` and `comm`, meaning that the infix operators "`and`" and "`or`" obey associativity and commutativity axioms:

```
fmod BOOL is
  sort Bool .
  ops true false : -> Bool .
  ops _and_ _or_ : Bool Bool -> Bool [assoc comm] .
  vars X Y : Bool .
  eq X and true = X [variant] .
  eq X and false = false [variant] .
  eq X or true = true [variant] .
  eq X or false = X [variant] .
endfm
```

There are five most general variants modulo AC for "X and Y", which are $\{(id, \texttt{X and Y}), (\{\texttt{X} \mapsto \texttt{true}\}, \texttt{Y}), (\{\texttt{Y} \mapsto \texttt{true}\}, \texttt{X}), (\{\texttt{X} \mapsto \texttt{false}\}, \texttt{false}), (\{\texttt{Y} \mapsto \texttt{false}\}, \texttt{false})\}$. Similarly, there are five most general variants for "X or Y".

An equational theory has the FVP (or it is called a *finite variant theory*) iff there is a finite and complete set of most general variants for each term. The theory `BOOL` is FVP, whereas the theory `NAT-VARIANT` does not have the FVP since there is an infinite number of variants in `NAT-VARIANT` for the term `X:Nat + 0`. It is generally undecidable whether an equational theory has the FVP (Bouchard *et al.* 2013); a semi-decision procedure is given by Meseguer (2015) that works well in practice and another technique based on the dependency pair framework is given by Escobar *et al.* (2012). The procedure by Meseguer (2015) works by computing the variants of all flat terms $f(X_1, \ldots, X_n)$ for any n-ary operator f in the theory and pairwise-distinct variables $X_1, \ldots, X_n$ (of the corresponding sort); the theory does have the FVP iff there is a finite number of most general variants for every such term (Meseguer 2015).

At the practical level, variants are generated by using an efficient narrowing strategy called the *(folding) variant narrowing strategy*, which was proved to be

complete for variant generation in Escobar *et al.* (2012) and terminates for all inputs provided that the theory has the FVP. Variant narrowing derivations correspond to narrowing sequences $t_0 \overset{\sigma_0}{\leadsto}_{e_0,Ax} t_1 \overset{\sigma_1}{\leadsto}_{e_1,Ax} \ldots \overset{\sigma_{n-1}}{\leadsto}_{e_{n-1},Ax} t_n$, where $t \overset{\sigma}{\leadsto}_{e,Ax} t'$ denotes a transition (modulo axioms in Ax) from term t to t' via the *variant equation e* (i.e., an equation e that is enabled to be used for narrowing thanks to the attribute `variant`) using the *equational unifier* σ. Assuming that the initial term t is normalized, each single transition $t \overset{\sigma}{\leadsto}_{e,Ax} t'$ (or variant narrowing step) is followed by the simplification of the term into its normal form by using all equations in the theory, which may include not only the variant equations in the theory but also non-variant equations (e.g., built-in equations in Maude). The composition $\sigma_0 \sigma_1 \sigma_{n-1}$ of all the unifiers along a narrowing sequence leading to t_n (restricted to the variables of t_0) is the *computed variant substitution* of this sequence. The *folding* refinement of variant narrowing that is implemented in Maude essentially consists in "folding", by subsumption modulo Ax, the narrowing tree for (E, Ax), which can in practice result in a finite narrowing graph that symbolically summarizes the, in general infinite, (E, Ax)-narrowing tree.

Maude provides the following command for variant generation:

```
get variants [ n ] in ⟨ModId⟩ : ⟨Term⟩ ,
```

where n is an optional argument that indicates the number of variants requested and $\langle ModId \rangle$ is the module where the command is run. There is also a meta-level command for variant generation, see Clavel *et al.* (2016).

Example 3
Consider the following equational theory (Clavel *et al.* 2016) for the *exclusive-or* symbol _*_ (i.e., an exclusive union operator _*_ for sets of natural numbers, `NatSet`, such that $X1 * X2$ is the set of natural numbers appearing in $X1$ or $X2$, but not both), where `mt` is the (empty set) identity element. Note that the notation `[NatSet]` denotes the *kind* of sort `NatSet` that, in addition to normal data of sort `NatSet`, can also contain "error expressions".

```
fmod EXCLUSIVE-OR is
  sorts Nat NatSet .  subsort Nat < NatSet .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [assoc comm] .
  vars X Z : [NatSet] .
  eq [idem] :     X * X = mt     [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] :       X * mt = X     [variant] .
endfm
```

We can check that the above theory has the FVP by asking Maude to generate all variants for the exclusive-or symbol * in the `EXCLUSIVE-OR` module, which delivers seven variants:

```
Maude> get variants in EXCLUSIVE-OR : X * Y .
Variant 1        Variant 2              Variant 7
[NatSet]: #1:[NatSet] * #2:[NatSet]  NatSet: mt    ...... [NatSet]: %1:[NatSet]
X --> #1:[NatSet]                    X --> #1:[NatSet]    X --> %1:[NatSet]
Y --> #2:[NatSet]                    Y --> #1:[NatSet]    Y --> mt
```

Observe that Maude can introduce fresh variables of two classes: (#*n*:*Sort*) or (%*n*:*Sort*). This is because it distinguishes between variables that are generated by the built-in unification modulo axioms (#*n*:*Sort*) and variables that are generated by variant-based unification or variant generation (%*n*:*Sort*) (Clavel *et al.* 2016). Also, note that the canonical form for any other instance of the term X * Y is subsumed modulo the axioms by one of the seven computed variants. For instance, when the substitution[2] {X ↦ 0 * s(0), Y ↦ 0 * s(0)} is applied to X * Y, the canonical form is just mt, but this is an instance of Variant 2. This is because the application (modulo associativity and commutativity of *) of equation [idem] causes 0 and s(0) to be cancelled. Other examples of variant generation can be found in the Maude manual (Clavel *et al.* 2016). Maude can also be asked to return the sequence of most general variants *incrementally*, which can be useful when a theory does not have the FVP. For instance, the term X:Nat + s(0) has an infinite number of most general variants in the theory of the module NAT-VARIANT. In such a case, Maude can either output the infinite sequence of variants to the screen (and the user can stop the process whenever she wants), or be given a bound *n* so that it generates only a maximum of *n* variants.

```
Maude> get variants [10] in NAT-VARIANT : X + s(0) .
Variant 1                      Variant 10
Nat: #1:Nat + s(0)    .........................    Nat: s(s(s(s(s(0)))))
X --> #1:Nat                   X --> s(s(s(s(0))))
```

In the case when the bound *n* is reached, the user can incrementally increase the bound so that the FVP is proved whenever the number of computed variants is smaller than the given bound. Unfortunately, the variant generation process can be infinitely repeated if the FVP does not hold.

In the following section, we show how proving that a theory has the FVP is much easier and fruitful by using GLINTS. Actually, we might even know that the FVP is not fulfilled and yet be interested in exploring the variant narrowing computation space of a number of terms in order to gain insights on how to modify the theory so that the FVP holds.

3 Folding variant narrowing trees in GLINTS: a running example

Let us consider again the equational specification for the exclusive-or theory above. This theory has the FVP since only seven most general variants exist for the symbol _*_. However, one might be interested to grasp why this specification fulfills the FVP, whereas slightly modified specifications of the exclusive-or theory are known to fail.

Example 4
Assume that we test the FVP after replacing the variable declaration X:[NatSet] of the original specification with X:Nat:

---

[2] Note that 0 * s(0) is of sort NatSet because Nat < NatSet.

Fig. 1. The FVP test for the modified non-FVP exclusive-or theory.

```
fmod EXCLUSIVE-OR-NOFVP is
  sorts Nat NatSet .  subsort Nat < NatSet .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [assoc comm] .
  var X : Nat . var Z : [NatSet] .
  eq [idem] :      X * X = mt     [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] :        X * mt = X     [variant] .
endfm
```

The variant generation process in Figure 1 is stopped after computing 43 variants for symbol `_*_` due to timeout, hence the result of the FVP test is *uncertain* yet this theory is known not to satisfy the FVP. One could investigate why this simple modification destroys FVP by inspecting the folding variant narrowing tree for the expression `X:[NatSet] * Y:[NatSet]` shown in Figure 2.

GLINTS can generate the folding variant narrowing tree of a given term in three ways: (i) stepwisely, by (manually) selecting a down triangle symbol ▼ that is shown below each narrowable node of the tree (see Figure 2); (ii) automatically until a fixed depth bound is reached; or (iii) automatically by using the more sophisticated control mechanism called *(equational) homeomorphic embedding* that is commonly used to ensure termination of unfolding-based program transformation and other symbolic methods (Leuschel 2002; Alpuente *et al.* 2017). Informally, a term t' embeds[3] another term t, in symbols $t \trianglelefteq t'$, if t (or a term that is equal to t modulo Ax) can be obtained from t' by deleting some symbols of t'; e.g., `s(s(X+Y)*(s(X)+Y))` embeds `s(Y*(X+Y))`, assuming commutativity of the `_*_` symbol. Nodes in the folding variant narrowing tree that embed a previous node in the same branch of the tree are highlighted in green and are decorated with symbol $\trianglelefteq$ below the node, as shown in Figure 2 (by clicking on the symbol, its closest embedded ancestor gets also highlighted).

In Figure 2, note that we have interactively produced variants up to V_{10} and could continue generating variants indefinitely, whereas the folding variant narrowing tree

---

[3] The order-sorted extension of homeomorphic embedding modulo equational axioms, such as associativity, commutativity, and identity that we use for Maude can be found in Alpuente *et al.* (2017).
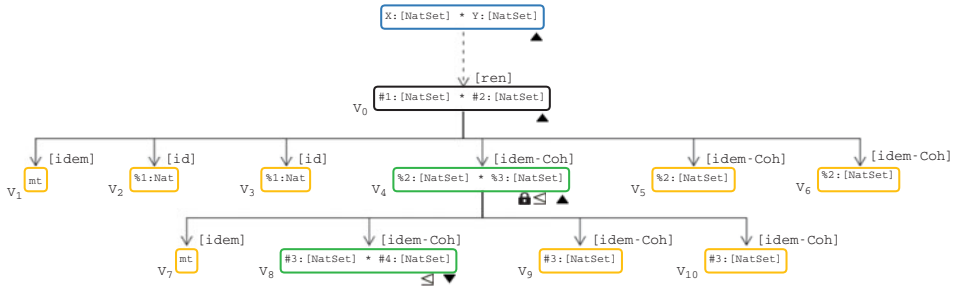
Fig. 2. Inspecting variant computations of the modified non-FVP exclusive-or theory.

Fig. 3. Comparison of nodes V_4 and V_8.

for the original EXCLUSIVE-OR theory stops at node V_6. Also, note that some potential narrowing steps stemming from the nodes of Figure 2 are not produced by the folding variant narrowing strategy as it avoids expanding nodes that are subsumed by previous ones. For instance, for node V_4, folding variant narrowing does not compute any children nodes equivalent to children V_2 and V_3 of node V_0. However, the theory EXCLUSIVE-OR-NOFVP does not have the FVP because nodes V_7, V_8, V_9, V_{10} are not subsumed by their counterpart nodes V_1, V_4, V_5, V_6, respectively, whereas they are subsumed for the theory EXCLUSIVE-OR, yielding the seven variants $V_0, \ldots, V_6$.

The fact that GLINTS automatically detects that node V_0 in Figure 2 is trivially *embedded* into node V_4, and that V_4 is *embedded* into node V_8 (actually they are all equal modulo variable renaming), warns about potentially infinite narrowing computations stemming from V_0 (it is said that $\trianglelefteq$ *whistles* (Leuschel 2002)). However, note that node V_8 is not a variant of V_4 (nor V_0). By comparing nodes V_4 and V_8 (enabled by pressing *Compare nodes* in the top-right menu), we obtain the information of Figure 3, which reveals that, even though V_4 and V_8 are equal modulo renaming, the computed variant substitutions are different.

After considering a negative example where GLINTS could help you to understand when and why the FVP of a theory can be lost after some changes, let us now analyze a positive example where an equational specification can satisfy the FVP after some changes.

Fig. 4. The FVP test for the newly modified exclusive-or theory with true verdict and variants for _*_.

Example 5
If we make the original specification and make the `_*_` symbol be associative, commutative, and with identity empty set element `mt`, then the theory does have FVP. This is shown in Figure 4; the list of computed variants for the operator `_*_` symbol is also shown, which has been retrieved by simply clicking on the corresponding 🔎 symbol of the right column.

```
fmod EXCLUSIVE-OR-ACU is
  sorts Nat NeNatSet NatSet .  subsort Nat < NeNatSet < NatSet.
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op mt : -> NatSet .
  op _*_ : NatSet NatSet -> NatSet [assoc comm id: mt] .
  op _*_ : NeNatSet NatSet -> NeNatSet [assoc comm id: mt] .
  var X : NeNatSet .    var Z : [NatSet] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
endfm
```

Note that this new specification relies on a subsort relation between sets of natural numbers (sort `NatSet`) and non-empty sets of natural numbers (sort `NeNatSet`), and it is simpler than the previous one because only one equation is needed.

If variable `X` were given, the sort `Nat` instead of `NeNatSet`, the mutated theory would not satisfy the FVP, as the reader could easily verify in GLINTS.

Folding variant narrowing trees can also be checked in GLINTS for the *(equational) closedness* property, which naturally extends to order-sorted equational theories (being executed by folding variant narrowing), the standard notion of closedness[4] of program calls that is used in the partial deduction of logic programs, meaning that the call is an instance of one of the specialized expressions. GLINTS implements the equational closedness check for the nodes of the deployed folding variant narrowing tree w.r.t. the root of the tree; this transfers to our setting the idea of *regularity* of a symbolic computation (in the terminology of Alpuente *et al.* (1998a) and Pettorossi and Proietti (1996)). Informally, a node in the tree is

---

[4] This notion was generalized to the narrowing-driven partial evaluation of functional-logic programs that are modeled as (unsorted) term rewriting systems (Alpuente *et al.* 1997; Alpuente *et al.* 1998b).

equationally closed (w.r.t. the tree root) if each narrowing redex in the term is an *equational instance* of the root node of the tree. For instance, for a tree with root (X*Y) and with one leaf node (0*X*Z), and assuming associativity of _*_, there are three redexes (namely, (0*X), (X*Z), and (0*X*Z)) and the leaf is closed. Note that neither closedness implies embedding nor the opposite: (0*0) is closed w.r.t. (X*Y) yet it does not embed it, and (0*X*Y) embeds (X*X) (because X embeds Y, for any variables X and Y) yet it is not closed w.r.t. it.

It is interesting to note that the notion of variant is closely related to the (functional logic) notion of *resultant* that is used in unfolding-based symbolic transformation techniques that rely on (some form of) narrowing, such as the narrowing-driven partial evaluator for TRSs of Alpuente *et al.* (1998b) and the partial evaluator Victoria for Maude equational theories of Alpuente *et al.* (2017), which is based on folding variant narrowing: given a narrowing tree for the term t in the equational theory $\mathscr{E}$, for each (*root-to-leaf*) narrowing derivation $t \leadsto_\sigma^* s$ in the tree, specialized (oriented) equations $t\sigma = s$ (also called *resultants*) can be extracted from the tree by piecing together the last term s of the narrowing derivation with the corresponding instance $t\sigma$ of the initial term t. Similarly to partial deduction, in the partial evaluator of Alpuente *et al.* (2017), *equational closedness* is the key property to ensure that, given a set Q of input expressions, the set resultants that can be extracted from a set of folding variant narrowing trees built in $\mathscr{E}$ for the terms of Q (each one as explained above) form a complete description that *correctly specializes* the original theory $\mathscr{E}$ to the considered set Q. In other words, all calls that may occur at run-time when any instance (modulo Ax) of a term of Q is executed in the specialized theory $\mathscr{S}$ are *covered* by $\mathscr{S}$ (i.e., folding variant narrowing computes the same solutions in $\mathscr{S}$ as in the original theory $\mathscr{E}$). This process is being efficiently implemented in Victoria thanks to the folding variant narrowing machinery developed in this work for GLINTS.

Similarly to the equational embedding test, the equational axioms and the order-sortedness information are both considered in the equational closedness test that is implemented in GLINTS. The tool checks this property automatically at any node in which the homeomorphic embedding whistles, and also when a node is interactively selected. It is signaled by an extra symbol 🔒 below the node (except for unnarrowable leaves, which are always trivially closed and are simply highlighted in orange). In Figure 2, all the nodes are equationally closed; actually, they are either unnarrowable or a syntactic instance of the tree root.

4 GLINTS at a glimpse

In this section, the main features of the graphical explorer GLINTS are outlined. Once a Maude module (or sequence of modules) has been input, the initial GLINTS panel allows (1) the folding variant narrowing space to be explored for a given term; and (2) the FVP to be checked (as explained in Section 3).

Running the graphical explorer and executing the corresponding textual narrowing commands of Maude is essentially the same regarding the processes that are conducted in the background (i.e., to some extent, the narrowing tree panel can

be interpreted as the visual correspondent of the `show-search-graph` command from the textual narrower). However, there is a dramatic difference in the tool output and in the thoroughness of the reasoning support provided by GLINTS.

4.1 Interactive tree unfolding and querying

Given an input term, the graphical narrowing tree panel initially contains two nodes: the input node and its normalized version w.r.t. the theory. Additions to the graph will be dictated by the user's exploration actions, which can be as follows.

Interactive exploration. GLINTS offers a graphical representation of the variant narrowing trees, including at each step (i) the narrowing redex, (ii) the applied variant equation, (iii) the equational unifier, and (iv) the computed variant substitution. GLINTS allows the narrowing tree to be easily navigated while providing thorough information regarding every node and edge in the tree. This is particular useful for a rich language such as Maude that supports sorts, subsorts, and overloading, and equational rewriting modulo axioms such as associativity, commutativity, and identity, where intuition is easily lost.

Each variant node is identified with a tag V_n, where n is the variant number assigned by Maude. When a node is selected (by a simple click), it is shaded in yellow so that the user can be constantly aware of the current selection. Node selection is useful for centering the node inside the tree layout and is also used for checking the equational closedness property. Fully detailed information about each variant can be displayed by double clicking on the corresponding node. Multiple variant information windows can be opened without updating the current tree.

As is common in visualization tools, the search trees can be scaled and subtrees can be hidden. This is done by pressing the ▲ symbol that is displayed below each node. By doing so, the entire (sub-)tree (except for its root) is removed from the displayed view of the tree. Taking into account that the size of the tree can become considerably large, zooming capabilities are also enabled.

Tree querying. A querying box is displayed at the bottom of the narrowing tree panel that allows information of interest to be easily searched in huge narrowing trees by undertaking a query that specifies a template for the search. A query is a filtering pattern with wildcards that define irrelevant symbols by means of the underscore character (_) and define relevant symbols by means of the question mark character (?). For instance, asking the query "_ * ?" in the tree of Figure 2 highlights expressions `#2:[NatSet]`, `%3:[NatSet]`, and `#4:[NatSet]` in nodes V_0, V_4, and V_8, respectively, as shown in Figure 5.

4.2 Automated tree unfolding, enriched views, and exporting

By using GLINTS, variant generation can be easily automated in multiple ways. Specifically, the user can ask the searcher to do one of the following: (i) deliver the first n variants of the considered initial term, (ii) compute the entire narrowing tree up to a given depth, or (iii) compute the entire narrowing tree until the embedding
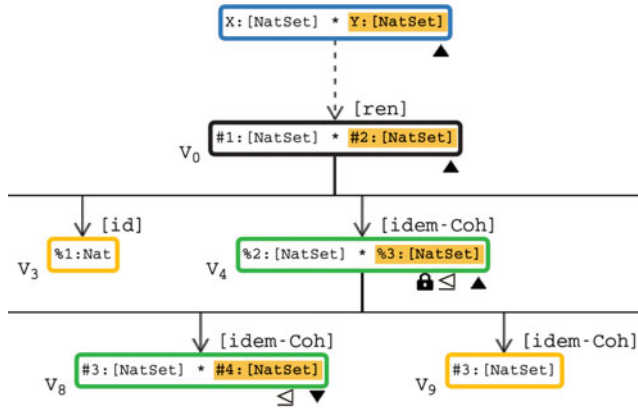
Fig. 5. Result of the query "_ * ?" for the VNT of the non-FVP exclusive-or theory.

whistles along all branches. In all cases, exploration of the tree stops whenever the corresponding termination criterion is met, namely (i) no more variants exist, (ii) bound is reached, (iii) embedding whistles, or (iv) timeout is surpassed.

By clicking the ≡ symbol that appears in the right corner of the window, a command menu is displayed that automates these capabilities by means of the following accessible buttons.

Depth-*k *(resp. *N-variants*) expansion. It unfolds the tree automatically down to its depth-k frontier (resp. until the nth variant has been computed). An input box allows one to fix the desired *upper* bound in the depth of the tree or in the number of solutions.

Embedding-based expansion. It automatically unfolds the variant narrowing tree by relying on equational homeomorphic embedding to ensuring finiteness. Roughly speaking, whenever a new node t_{n+1} is to be added to a branch, GLINTS checks whether t_{n+1} embeds any of the terms already in the sequence. If that is the case, potential non-termination is detected and the computation is stopped. Otherwise, t_{n+1} is safely added to the branch and the computation proceeds.

The key to successfully debugging complex applications is to restrict the displayed information to sensitive parts of the tree. In GLINTS, it is possible to tune the information displayed by the explorer by using enriched views and reporting facilities as follows.

Enriched views. GLINTS supports two distinct views, namely the standard view and the instrumented view. The standard view (which is the default mode of GLINTS) focuses on the narrowing steps, whereas the instrumented view completes the picture with all the internal reduction steps that are performed up to reaching the canonical form of each variant. That is, the instrumented view reaps every single application of an equation, algebraic axiom, or built-in operation. This view is enabled by pressing the button *Show normalization*. The options to show/hide the equation labels and to show/hide the unifiers that enable each narrowing step of the tree (restricted to the
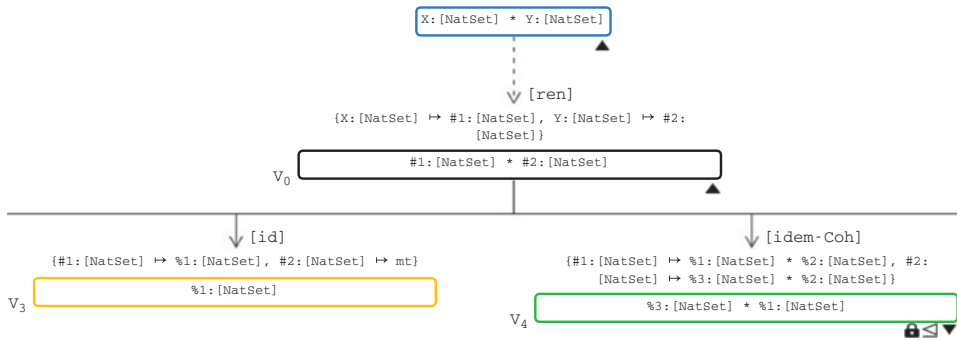
Fig. 6. Enriched view showing equational unifiers for the original exclusive-or theory (fragment).

variables of the term, as shown in Figure 6) are also available by two corresponding buttons.

Comparing and exporting. Given the currently deployed narrowing tree, the complete list of computed variants can be shown and exported by clicking the option *Export variants*. In order to easily discern the differences between two variants, a *Compare variants* button is also provided that confronts two variant nodes (selected by just two consecutive clicks) in a new window where they are displayed next to each other, one on the left half of the window and the other one on the right half. GLINTS can export both the entire narrowing tree or any of its branches in two different formats, namely as an object in JSON format and as a term in Maude's meta-level representation, both of which are suitable for automated processing. This allows other tools that use GLINTS for narrowing execution to implement their own analysis on the trees delivered by GLINTS. The meta-representation of terms can be visually displayed, which is particularly useful for the analysis of object-oriented computations where object attributes can only be unambiguously visualized in the meta-level (desugared) terms.

A starting guide that contains a complete description of all the settings and detailed sessions can be found at `http://safe-tools.dsic.upv.es/glints/download/quickstart.pdf`.

5 Implementation

In this section, we discuss some relevant implementation details of the variant explorer GLINTS.

5.1 Architecture of GLINTS

GLINTS has the classical architecture of a web application, which consists of two main components (the front-end and the back-end), as depicted in Figure 7. The two components are connected via a JSP-based layer that is implemented in Java (450 lines of Java source code). The front-end (or presentation layer) consists of
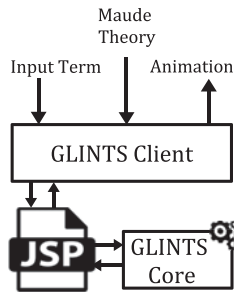
Fig. 7. Architecture of GLINTS.

3K lines of Javascript, HTML5, and CSS source code, and provides GLINTS with an intuitive Web user interface. The back-end (or core engine) supports GLINTS services and consists of 200 function definitions (2K lines of Maude source code).

5.2 Extending Maude's variant meta-operations

One of the main challenges in the implementation of a trace-based Maude tool such as GLINTS is to make explicit the concrete sequence of internal term transformations occurring in a specific Maude computation, which is generally hidden and inaccessible within Maude's rewriting and narrowing machineries. For the case of variant narrowing computation traces, the basic information that is necessary to visually deploy the variant narrowing trees can be essentially obtained by invoking the `metaGetVariant` and `metaGetIrredundantVariant` meta-operations. That is the only way to retrieve the precise information that makes the structure of the tree explicit. Specifically, what Maude outputs is the following (in this order): (i) the computed variant term, (ii) the computed variant substitution, (iii) the largest index n of any fresh variable appearing in the solutions, (iv) the identifier of the parent variant, and (v) a boolean flag that indicates whether or not there are more variants in the current tree level.

However, for the sake of efficiency, other relevant information that is key for variant narrowing debugging and understanding is not disclosed by Maude, either at the meta-level (as returned by the `metaGetVariant` and `metaGetIrredundantVariant` operations themselves) or at the source-level (as delivered in raw text format by the standard Maude interactive debugger, which furthermore cannot be manipulated as a meta-level expression by Maude). To provide the user with a deeper and more agile debugging experience, we have enriched the highly efficient developer version of Maude that we implemented in previous work, Mau-Dev[5] (Alpuente *et al.* 2016; Mau-Dev 2016), with two new meta-operations, namely `metaGetVariantsExt` and `metaGetIrredundantVariantExt` that have been implemented in C++. By doing this, besides piecing everything together and giving a graphical reconstruction of

---

[5] Mau-Dev has been developed under the GPLv2 license (which is the one enforced by Maude) and is fully compatible with Maude while preserving the efficiency of all standard (meta-level) operations and commands.

Table 1. Execution results of the `metaGetVariant` and `metaGetVariantExt` operations

	Number of variants	metaGetVariant		metaGetVariantExt	
		size (kB)	time (s)	size (kB)	time (s)
Exclusive-or	40	7.37	2.49	12.34	2.48
	45	8.81	24.82	14.42	24.56
	50	10.37	302.18	16.62	299.29
Fibonacci	40	520.23	3.51	1,417.26	3.59
	45	2,198.07	20.52	5,151.39	20.94
	50	5,751.55	406.59	15,675.13	415.14
Flip-graph	500	4,804.66	3.05	7,259.92	3.09
	1,000	19,520.91	30.33	29,387.01	30.93
	2,000	80,372.41	360.29	120,769.01	361.54
Parser	2,500	1,961.51	3.91	3,067.46	3.92
	5,000	5,027.82	16.88	7,238.53	17.37
	10,000	13,178.03	81.64	17,598.87	81.99

the variant narrowing tree, GLINTS also distills the equations, axioms, and built-in operators applied in (simplification and) narrowing steps, together with the equational unifier that enables each step.

Table 1 provides some figures regarding the execution of the new `metaGetVariantExt` operation in comparison with the standard `metaGetVariant` operation. We have tested both implementations on a 3.3 GHz Intel Xeon E5-1660 with 64 GB of RAM by generating a number of variants for a collection of Maude programs that are all available at the GLINTS website: *Exclusive-or*, the classical specification of the boolean XOR; *Fibonacci*, a Maude specification that computes the Fibonacci sequence (Clavel *et al.* 2016); *Flip-graph*, a variant of the classical *flip* function for binary graphs (instead of trees) taken from Alpuente *et al.* (2017); and *Parser*, a generic parser for languages generated by simple, right regular grammars also from Alpuente *et al.* (2017). Specifically, for each Maude program, we have asked GLINTS to compute three different numbers of variants, which takes from a few seconds to a few minutes to generate. We have measured the `metaGetVariant` invocations on a statically compiled version of the last alpha release of Maude (alpha 111a), whereas the `metaGetVariantExt` invocations have been benchmarked on a Mau-Dev executable that is based on the same alpha version.

The two `size` columns correspond to the size (in kilobytes) of the generated narrowing tree (up to the requested variant), whereas the two `time` columns show the average of five different measures of the computation time (in seconds). As our experiments show, the incurred overheads w.r.t. the original meta-operation are almost negligible. Note that even for extremely huge narrowing trees, the amount of data handled is much higher w.r.t. the original meta-operation (with an average increasement factor of 1.8), yet the execution time is practically identical. Actually, some executions are even faster in the extended version (e.g., computing the 50th variant of the exclusive-or example), which can be explained by the known side-effects of Maude's garbage collector and cache

memory hits and misses. Further details and runnable code are available at
http://safe-tools.dsic.upv.es/glints/pages/experiments.jsp

6 Concluding remarks

Visualization of program executions has received much attention for program debugging, optimization, profiling, and understanding in symbolic execution frameworks such as (Concurrent) Constraint Logic Programming (Deransart *et al.* 2006). However, with the exception of GLINTS, no such visualization tool exists for variant narrowing computations in Maude, let alone one with the capability to reason about equational properties such as embedding and closedness modulo axioms and the FVP.

Besides the applications outlined in this article, further applications could benefit from variant generation in GLINTS. Actually, an important number of applications (and tools) are currently based on variant generation: for instance, the protocol analyzers Maude-NPA (Escobar *et al.* 2009) and Tamarin (Meier *et al.* 2013), proofs of coherence and local confluence (Durán and Meseguer 2012), termination provers (Durán *et al.* 2009), variant-based satisfiability checkers (Meseguer 2015), the partial evaluator Victoria (Alpuente *et al.* 2017), and different applications of symbolic reachability analyses (Bae *et al.* 2013). As an application example, protocol analysis tools that rely on variant computation could identify all the intermediate variant states that are associated to a concrete protocol state and how one is generated from the other (which is convoluted in the output provided by Maude), thereby allowing deep optimizations to cut down the search space. Indeed, many protocol analysis tools suffer from huge memory problems due to complex equational theories that generate lots of variants.

As future work, we plan to address several extensions of GLINTS, such as computing constructor variants (Meseguer 2015) and irredundant variants (Clavel *et al.* 2016), and supporting irreducibility constraints (Erbatur *et al.* 2012).

References

ALPUENTE, M., BALLIS, D., FRECHINA, F. AND SAPIÑA, J. 2016. Assertion-based analysis via slicing with ABETS. *Theory and Practice of Logic Programming 16,* 5–6, 515–532.

ALPUENTE, M., CUENCA-ORTEGA, A., ESCOBAR, S. AND MESEGUER, J. 2017. Partial evaluation of order-sorted equational programs modulo axioms. In *Proc. of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, Lecture Notes in Computer Science, vol. 10184. Springer, 3–20.

ALPUENTE, M., FALASCHI, M., MORENO, G. AND VIDAL, G. 1997. Safe folding/unfolding with conditional narrowing. In *Proc. of the 6th International Joint Conference on Algebraic and Logic Programming (ALP 1997)*. Lecture Notes in Computer Science, vol. 1298. Springer, 1–15.

ALPUENTE, M., FALASCHI, M. AND VIDAL, G. 1998a. A unifying view of functional and logic program specialization. *ACM Computing Surveys 30,* 3es, 9es.

ALPUENTE, M., FALASCHI, M. AND VIDAL, G. 1998b. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems 20,* 4, 768–844.

Bae, K., Escobar, S. and Meseguer, J. 2013. Abstract logical model checking of infinite-state systems using narrowing. In *Proc. of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, LIPIcs, vol. 21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 81–96.

Bouchard, C., Gero, K. A., Lynch, C. and Narendran, P. 2013. On forward closure and the finite variant property. In *Proc. of the 9th International Symposium on Frontiers of Combining Systems (FroCos 2013)*. Lecture Notes in Computer Science, vol. 8152. Springer, 327–342.

Chen, W. and Warren, D. S. 1996. Tabled evaluation with delaying for general logic programs. *Journal of the ACM 43,* 1, 20–74.

Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. and Talcott, C. 2016 *Maude Manual (Version 2.7.1)*. Technical Report, SRI International Computer Science Lab.

Comon-Lundh, H. and Delaune, S. 2005. The finite variant property: How to get Rid of some algebraic properties. In *Proc. of the 16th International Conference on Rewriting Techniques and Applications (RTA 2005)*. Lecture Notes in Computer Science, vol. 3467. Springer, 294–307.

Deransart, P., Hermenegildo, M. V. and Maluszynski, J. 2006 *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*. Lecture Notes in Computer Science, vol. 1870 Springer.

Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J. and Talcott, C. 2016. Built-in variant generation and unification, and their applications in maude 2.7. In *Proc. of the 8th International Joint Conference on Automated Reasoning (IJCAR 2016)*. Lecture Notes in Computer Science, vol. 9706. Springer, 183–192.

Durán, F., Lucas, S. and Meseguer, J. 2009. Termination modulo combinations of equational theories. In *Proc. of the 7th International Symposium on Frontiers of Combining Systems (FroCos 2009)*. Lecture Notes in Computer Science, vol. 5749. Springer, 246–262.

Durán, F. and Meseguer, J. 2012. On the Church–Rosser and coherence properties of conditional order-sorted rewrite theories. *The Journal of Logic and Algebraic Programming 81,* 7–8, 816–850.

Erbatur, S., Escobar, S., Kapur, D., Liu, Z., Lynch, C., Meadows, C., Meseguer, J., Narendran, P., Santiago, S. and Sasse, R. 2012. Effective symbolic protocol analysis via equational irreducibility conditions. In *Proc. of the 17th European Symposium on Research in Computer Security (ESORICS 2012)*. Lecture Notes in Computer Science, vol. 7459. Springer, 73–90.

Escobar, S., Meadows, C. and Meseguer, J. 2009. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V (FOSAD 2007/2008/2009 Tutorial Lectures)*. Lecture Notes in Computer Science, vol. 5705. Springer, 1–50.

Escobar, S., Sasse, R. and Meseguer, J. 2012. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming 81,* 7–8, 898–928.

Hanus, M. 2013. Functional logic programming: From theory to curry. In *Programming Logics. Essays in Memory of Harald Ganzinger*. Lecture Notes in Computer Science, vol. 7797. Springer, 123–168.

Leuschel, M. 2002. Homeomorphic embedding for online termination of symbolic methods. In *The Essence of Computation. Essays Dedicated to Neil D. Jones on the Occasion of his 60th Birthday*. Lecture Notes in Computer Science, vol. 2566. Springer, 379–403.

Mau-Dev 2016. The `Mau-Dev` Web site. URL: `http://safe-tools.dsic.upv.es/maudev`. [Accessed on July 2017].

MEIER, S., SCHMIDT, B., CREMERS, C. AND BASIN, D. A. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. of the 25th International Conference on Computer Aided Verification (CAV 2013)*. Lecture Notes in Computer Science, vol. 8044. Springer, 696–701.

MESEGUER, J. 1992. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science 96,* 1, 73–155.

MESEGUER, J. 2015. Variant-based satisfiability in initial algebras. In *Proc. of the 4th International Workshop for Safety-Critical Systems (FTSCS 2015)*. Communications in Computer and Information Science, vol. 596. Springer, 3–34.

MESEGUER, T. 2006. From OBJ to Maude and beyond. In *Proc. of Algebra, Meaning, and Computation. Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. Lecture Notes in Computer Science, vol. 4060. Springer, 252–280.

PETTOROSSI, A. AND PROIETTI, M. 1996. A comparative revisitation of some program transformation techniques. In *Proc. of International Dagstuhl Seminar on Partial Evaluation*. Lecture Notes in Computer Science, vol. 1110. Springer, 355–385.

YANG, F., ESCOBAR, S., MEADOWS, C., MESEGUER, J. AND NARENDRAN, P. 2011. Theories of homomorphic encryption, unification, and the finite variant property. In *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*. ACM Press, 123–133.