UNIVERSITAT POLITÈCNICA DE VALÈNCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
DOCTORADO EN INFORMÁTICA

Partial Evaluation of Equational Logic Theories

PHD THESIS

Presented by:
Angel Cuenca Ortega

Supervisors:
María Alpuente Frasnedo
Santiago Escobar Román

Valencia, September 2019

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Partial Evaluation of Equational Logic Theories

A dissertation submitted by Angel Cuenca Ortega in fulfillment for the degree of Doctor of Philosophy in Computer Science at the Universitat Politècnica de València

Valencia, September 2019

Partial Evaluation of Equational Logic Theories

Author:
Angel Cuenca Ortega

Supervisors:
María Alpuente Frasnedo Universitat Politècnica de València
Santiago Escobar Román Universitat Politècnica de València

External Evaluators:
Mateu Villaret Auselle Universidad de Girona
Ginés Moreno Valverde Universidad de Castilla-La Mancha
Pascual Julián Iranzo Universidad de Castilla-La Mancha

Jury:
Silvia Abrahao Gonzales Universitat Politècnica de València (President)
Mateu Villaret Auselle Universidad de Girona (Secretary)
Marisa Navarro Gómez Universidad del País Vasco (Vocal)

Valencia, September 2019

This work has been supported by the SENESCYT, Ecuador (scholarship program 2013)

“Work gives you meaning and purpose and life is empty without it.”
“El trabajo te da sentido y propósito, y la vida está vacía sin él."

Stephen Hawking

A mi madre, hermanos y hermana, sobrinos y sobrinas, y amigos
por su apoyo incondicional.

To my mother, brothers and sister, nephews and nieces, and friends
for their unconditional support.

Abstract

Partial evaluation is a powerful and general program optimization technique that preserves program semantics and has many successful applications. Optimization is achieved by specializing programs w.r.t. a part of their input data so that, when the residual or specialized program is executed on the remaining input data, it produces the same outcome than the original program with all of its input data. Existing PE schemes do not apply to expressive rule-based languages like Maude, CafeOBJ, OBJ, ASF+SDF, and ELAN, which support: 1) rich type structures with sorts, subsorts, and overloading; and 2) equational rewriting modulo various combinations of axioms such as associativity, commutativity, and identity. This thesis develops the foundations of equational order sorted partial evaluation and illustrates the key concepts by showing how they apply to the specialization of expressive programs written in Maude. Our partial evaluation scheme is based on an automatic unfolding algorithm that computes term *variants* and relies on high-performance *order-sorted equational least general generalization* and *order-sorted equational homeomorphic embedding* algorithms for ensuring termination. We show that our partial evaluation technique is sound and complete for order-sorted equational theories that may contain various combinations of associativity, commutativity, and/or identity axioms for different binary operators. Finally, we present Victoria, the first partial evaluator for Maude's order-sorted equational theories, and demonstrate the effectiveness of our partial evaluation scheme on several examples where it shows significant speed-up.

Resumen

La evaluación parcial de programas es una técnica general y potente de optimización de programas que preserva su semántica y tiene muchas aplicaciones relevantes. La optimización se consigue al especializar programas con respecto a una parte de sus datos de entrada, lo que produce un nuevo programa llamado residual o programa especializado tal que, al ejecutarlo con los datos de entrada restantes, producirá el mismo resultado que produce el programa original con todos sus datos de entrada. Los esquemas de evaluación parcial existentes no son aplicables a lenguajes expresivos basados en reglas como Maude, CafeOBJ, OBJ, ASF+SDF y ELAN, los cuales soportan: 1) sofisticados tipos estructurados con subtipos y sobrecarga de operadores; y 2) teorías ecuacionales modulo varias combinaciones de axiomas tales como asociatividad, conmutatividad e identidad. Esta tesis desarrolla las bases teóricas necesarias e ilustra los conceptos principales para su aplicación a programas expresivos escritos en el lenguaje Maude. El esquema de evaluación parcial presentado en esta tesis está basado en un algoritmo automático de desplegado que computa *variantes* de términos. Para asegurar la terminación del proceso de especialización se han diseñado algoritmos de alto rendimiento para la *generalización ecuacional menos general con tipos ordenados* y *subsunción homeomórfica ecuacional con tipos ordenados*. Se muestra que la técnica de evaluación parcial desarrollada es correcta y completa para teorías de reescritura convergentes que pueden contener varias combinaciones de axiomas de asociatividad, conmutatividad y/o identidad para diferentes operadores binarios. Finalmente se presenta Victoria, el primer evaluador parcial para teorías ecuacionales de tipos ordenados para el lenguaje Maude, y se demuestra la efectividad y el incremento en eficiencia ganado a través de experimentos realizados con ejemplos reales.

Resum

L'avaluació parcial de programes és una tècnica general i potent d'optimització de programes que preserva la seua semàntica i té moltes aplicacions rellevants. L'optimització s'aconseguix a l'especialitzar programes respecte a una part de les seues dades d'entrada, la qual cosa produïx un nou programa cridat residual o programa especialitzat tal que, a l'executar-ho amb les dades d'entrada restants, produirà el mateix resultat que produïx el programa original amb totes les seues dades d'entrada. Els esquemes d'avaluació parcial existents no són aplicables a llenguatges expressius basats en regles com Maude, CafeOBJ, OBJ, ASF+SDF i ELAN, els quals suporten: 1) sofisticats tipus estructurats amb subtipus i sobrecàrrega d'operadors; i 2) teories equacionals mòdul diverses combinacions d'axiomes com asociativitat, conmutativitat i identitat. Esta tesi desenrotlla les bases teòriques necessàries i il·lustra els conceptes principals per a la seua aplicació a programes expressius escrits en el llenguatge Maude. L'esquema d'avaluació parcial presentat en esta tesi està basat en un algoritme automàtic de desplegat que computa *variants* de termes. Per a assegurar la terminació del procés d'especialització s'han dissenyat algoritmes d'alt rendiment per a la *generalització ecuacional menys general amb subtipus ordenats* i *subsunción ecuacional homeomórfica amb subtipus ordenats*. Es mostra que la tècnica d'avaluació parcial desenrotllada és correcta i completa per a teories de reescriptura convergents que poden contindre diverses combinacions d'axiomes d'asociativitat, conmutativitat i identitat per a diferents operadors binaris. Finalment es presenta Victoria, el primer avaluador parcial per a teories equacionals de tipus ordenats per al llenguatge Maude i es demostra l'efectivitat i l'increment en eficiència guanyat a través d'experiments realitzats amb exemples reals.

Acknowledgement

Many people have contributed at this stage of my life, without them I would not have been able to finish this thesis.

First of all I would like to thank my supervisors, María and Santiago, for their guidance and dedication during all these years. Without a doubt, I think there are no better supervisors than them. They always find a space in their busy schedules to help their students and share their knowledge. Their passion for research makes them an example for many young researchers. In truth, I feel very happy and fortunate to have been part of the ELP Group and, much more, to have done my thesis under the supervision of María and Santiago.

I am very grateful to José Meseguer with whom I have had the pleasure to collaborate these years. I can only feel gratitude to him for allowing me to learn from his enriched experience.

I also thank to all the members of the ELP Group, and particularly to my laboratory mates during all these years: Fernando, Lidia, David, Julia, and Danny.

I want to extend this gratitude to Demis Ballis for giving me the opportunity to work with him in my research stay at *Università degli Studi di Udine*.

Especial thanks to my lunch mates: Ana, José, Beatriz, Lenin, Nana, Sipan, Julio, Carlos, Jairo, and Xavier. All of them made me feel like in a family.

I cannot forget all my dear friends with whom I shared great moments: Priscas and her family, Juanjo, Joty and his family, Karlita and her family, Pepita and her family, Gaby and Oscar, Pablo and his family, Miguel and Gela, Alex and Doris, Patty, Carolina, William, D. César, Riccardo, Renaud, Luis, Jona and his family, Adrián, and Francisco.

Furthermore, a deep thanks to my closest friends who helped me in different ways: Miguel D., Joha O., Manuel, Luis V., Bismar, Alberto S., Paúl, Edith, Markus, Elías, Gaby C., Yoselyn, Jordy, BL., BC, Ronald, Jorge M., Jessica G., Max, Jhonny, Liston, Ivonne, Julio, Lorenzo, José G., Mario G, Bryan C., Carlos V., Fernando T., Fátima, Cristoffer, Abel, Filippo, Giuseppe, Alberto M., Emilio, and Daniele.

Finally, I extend my thanks to *SENESCYT* for the support provided for my studies. Also, I thank the *Universidad de Guayaquil* that is my place of work.

Angel Cuenca Ortega.
Valencia, September 2019

Contents

List of Figures

Chapter 1

Introduction

Specialization is Nature's Strategy for Winning.
Marcus Buckingham
GO, Put Your Strengths to Work, 2007

In an evolving world, the best way to survive is to show your strengths. You need those characteristics that make you differentiate and specialize. Specialize to win. Specialize *something* to achieve better results, without changing the objective of that *something*. This fantastic idea is applied in computer science to specialize programs, called *program specialization* or *partial evaluation*, where new correct and more efficient programs are obtained from a generic or more inefficient program version sometimes achieving speedups of several orders of magnitude.

Partial evaluation (PE) is an automatic program transformation technique for program optimization that preserves program semantics [Jones et al. 1993]. The program that undertakes the partial evaluation process is called a *partial evaluator*. The partial evaluation process relies on some stopping criterion to guarantee termination of the specialization process [Alpuente et al. 1998a, 2018]. The number of applications of partial evaluation is extremely extensive, e.g., in the fields of pattern recognition, neural network training, scientific computing, model-driven development, domain-specific language engineering, generic programming, and test-case generation, just to mention a few [Cadar and Sen 2013; Cook and Lämmel 2011; Jones et al. 1993].

Partial evaluation has proven to be useful in many programming paradigms. For instance, in the area of functional programming [Consel and Danvy 1993; Jones et al. 1993; Turchin 1986] and logic programming (also known as *partial deduction*) [Gallagher 1993; Komorowski 1982; Lloyd and Shepherdson 1991; Pettorossi and Proietti 1994]. Also for imperative languages such as C [Consel et al. 1998] and Java [Ji and Bubel 2012; Singh and King 2014], and in declarative languages such as Prolog [Leuschel and Bruynooghe 2002], Haskell [Jones 1990], and Curry [Albert et al. 1999; Alpuente et al. 1999; Hanus and Peemöller 2014].

1.1 Partial Evaluation

Partial evaluation is a source-to-source program transformation technique in which a program P is specialized to a part S of its input data I, with $I = (S, D)$. More precisely, given a program

P and *static* known input data S, the partial evaluator generates a new program P' (called the *residual program*), which is semantically equivalent to P when P is fed with data I. That is, given the remaining (unknown) input data D (also called *dynamic*), P' will produce the same outcome O that P produces with the input $I = (S,D)$, ensuring correctness of the transformation. Figure 1.1 shows the essence of a partial evaluator called *mix* by using the following visual notation: data values are enclosed in ellipses, programs are in boxes, and data programs are in both. Furthermore, continuous arrows are used for the partial evaluation process, while the dashed arrows feeds program with their inputs and delivers outcomes of their execution. The underlying idea of the PE process consists of: 1) performing as many computations as possible with the static input data and with the results of those computations and 2) producing the specialized program from the *resultants* of those computations. The partially evaluated resulting program is able to run with the remaining unknown input data. Thus, a partial evaluator performs a mixture of computation and code generation actions; this is the reason why PE was called *mixed computation* in [Ershov 1982].

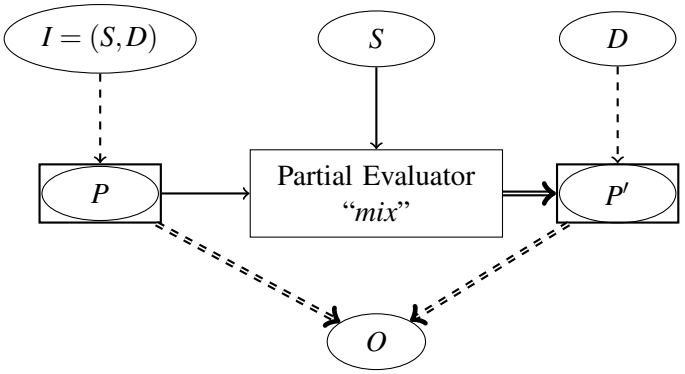


FIGURE 1.1: The essence of Partial Evaluation.

When a partial evaluator performs the corresponding computations using the static input data, we can say that there exists some *information propagation*. Partial evaluation aims to produce the best possible specialized program while maximizing the propagation of the information. Therefore, it is expected that the specialized program runs faster (on the remaining input data) than the original (on all of its input data), since it is possible to avoid some computations that are performed once (and for all) during the specialization process.

Example 1.1. *Consider a program P that defines the factorial and power functions on integer numbers:*

$$n! = \begin{cases} 1 & if\ n = 0 \\ n * (n-1)! & if\ n \geq 1 \end{cases} \qquad power(b,e) = \begin{cases} 1 & if\ e = 0 \\ b * power(b,e-1) & if\ e \geq 1 \end{cases}$$

We can specialize the program P w.r.t. the input call $power(X,2!)$*, with X being a variable. After applying the partial evaluation process, the resulting program is:*

$$power'(X) = X * X$$

where any calls to the factorial and power functions have been removed, while it is able to execute $power(X,2!)$ *for any given X.*

However, the main issues related to automatic PE concern the preservation of the (operational) semantics, the termination of the process, and the effectiveness of the transformation, i.e., attaining a significative execution speedup for a large class of programs [Sørensen et al. 1996].

Partial evaluation relies on well-known techniques from the literature on program transformation, such as the *folding* and *unfolding* transformations, which were first introduced by [Burstall and Darlington 1977] for functional programs and are exploited in different ways by the various PE techniques. Unfolding is essentially the replacement of a call by its definition, with appropriate substitutions. Folding is the inverse transformation, that is, the replacement of some piece of code by an equivalent function call. The partial evaluation of functional programs is usually based on unfolding expressions and constant propagation, while partial evaluation for logic languages exploit the power of unification for parameter propagation [Glück and Sørensen 1994]. The unfolding of functional logic programming languages is based on *narrowing* [Fay 1979; Slagle 1974], which is a combination of unification for parameter passing and term rewriting. This is the operational principle of integrated functional logic languages, and is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers).

1.1.1 Narrowing-driven Partial Evaluation

Narrowing-driven PE (NPE) [Alpuente et al. 1996, 1997a, 1998a,b] is a generic algorithm for the specialization of multiparadigm functional logic languages that are executed by narrowing. The NPE method generalizes the theoretical framework for the partial deduction of logic programs established in [Lloyd and Shepherdson 1991; Martens and Gallagher 1995] to functional logic programs, with the key concepts being extended to suitably cope with functions and nested function calls (e.g., the *closedness* condition that ensures that all calls that might occur during the execution of the specialized program are covered by the specialized program). NPE has better opportunities for optimization than the more standard PE of functional programs thanks to the combination of the functional dimension of narrowing with the power of logic variables and unification. The NPE scheme of [Alpuente et al. 1998a] is parametric w.r.t. an *unfolding rule* used to construct finite derivations for an expression and an *abstraction operator* used to guarantee that only finitely many expressions are evaluated. In the remaining of this section, we recall the key ideas of the NPE scheme [Alpuente et al. 1998a].

We consider a set R of rewrite rules and a set Q of program calls (i.e., input terms). The aim of NPE [Alpuente et al. 1998a] is to derive a new set R' of rules (called a partial evaluation of R w.r.t. Q, or a partial evaluation of Q in R) which computes the same answers and irreducible forms (w.r.t. narrowing) than R for any term t that is inductively covered (*closed*) by the calls in Q. This means that every subterm of the leaves of the execution tree for t using the rules R that can be narrowed (modulo associativity, commutativity, and identity axioms) in R can also be narrowed in R'. Roughly speaking, R' is obtained by first constructing a *finite* (possibly partial) narrowing tree for the input term t and then gathering together the set of *resultants* $t\theta_1 \to t_1, \ldots, t\theta_k \to t_k$ that can be constructed by considering the leaves of the tree, say $t_1, \ldots, t_k$, and the computed substitutions for the leaves, say $\theta_1, \ldots, \theta_k$ of the associated branches of the tree (i.e., a resultant rule $t\theta_i \to t_i$ is associated to each root-to-leaf derivation of the narrowing tree). Resultants perform what in fact is an n-step computation in R, with $n > 0$, by means of a single step computation in R'. The unfolding process is repeated for the set $\mathscr{L}$ of all narrowable subterms of $t_1, \ldots, t_k$ that are not covered by Q. In order to ensure that the gathered resultants form a complete description covering all calls that may occur at run-time in the final specialized

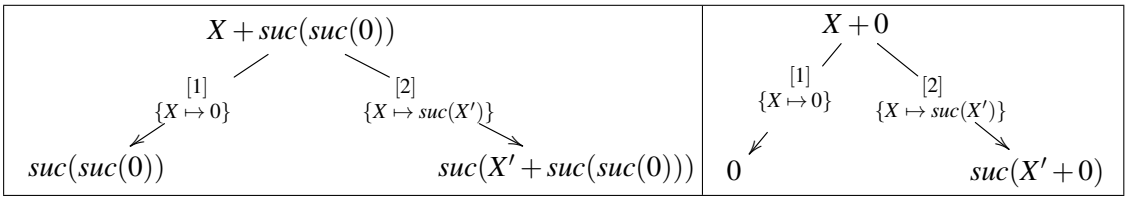
theory R', partial evaluation must rely on a parametric general notion of *Q-closedness* that is not a mere syntactic subsumption check, that every subterm occurring in the leaves of the tree(s) is a substitution instance of one of the terms being specialized. However, in order to properly deal with nested function calls, the closedness notion recurses over the structure of the subterm. Informally, a term t rooted by a defined function symbol is Q-closed w.r.t. R iff it is an instance of a term of Q by the matching substitution θ and the terms in θ are recursively closed by Q. For instance, given a function symbol $\bullet$, the term $t = a \bullet (Z \bullet a)$ is closed w.r.t. the call set $Q_1 = \{a \bullet X, Y \bullet a\}$ and also w.r.t. the call set $Q_2 = \{X \bullet Y\}$, but it is not closed w.r.t. $Q_3 = \{a \bullet X\}$. The following example illustrates the NPE method.

Example 1.2. *Consider the following theory for addition of natural numbers where* 0 *and suc are constructor symbols,* $+$ *is a defined symbol, and X and Y are variables:*

$$0 + Y \rightarrow Y \tag{1}$$

$$suc(X) + Y \rightarrow suc(X + Y) \tag{2}$$

Following the NPE approach, in order to specialize the program w.r.t. the goals $X + suc(suc(0))$ *and* $X + 0$*, we compute the narrowing trees depicted in Figure 1.2. Observe that, in both trees, all of the calls in their leaves are closed w.r.t. the tree root.*



(a) (b)

FIGURE 1.2: Narrowing trees for the goal $X + suc(suc(0))$ and $X + 0$.

The resulting programs for the given theory are shown in Figure 1.3:

$0 + suc(suc(0)) = suc(suc(0))$ $suc(X') + suc(suc(0)) = suc(X' + suc(suc(0)))$	$0 + 0 = 0$ $suc(X') + 0 = suc(X' + 0)$

(a) (b)

FIGURE 1.3: Specialized programs for the goals $X + suc(suc(0))$ and $X + 0$.

As mentioned before, the main issues related to automatic PE (besides performance improvement) concern both *termination* (i.e., given any input goal, partial evaluation should always reach a stage at which there is no way to continue) and (partial) *correctness* (i.e., the residual program behaves as the original one for the considered input terms, provided the PE process terminates).

As for termination, any partial evaluation algorithm deals with two classical termination problems: the so-called *local* termination problem (the termination of the unfolding, i.e., how to control that the deployment of the narrowing-trees is finite, which is managed by an unfolding rule), and the *global* termination problem (which concerns termination of repetitive unfolding, i.e., stopping the repetitive construction of narrowing-trees while still guaranteeing that the desired amount of specialization is retained and that the closedness condition is reached).

As for local termination, NPE [Alpuente et al. 1998a] relies on the notion of *homeomorphic embedding* ($\trianglelefteq$) for controlling the unfolding during the construction of the narrowing trees. As

for the global control, NPE relies on an *abstraction* operator which is based on the classical notion of *least general generalization (lgg)* that is used to compute a safe approximation A of $Q \cup \mathscr{L}$, as depicted below. Figure 1.4 illustrates the general NPE scheme.

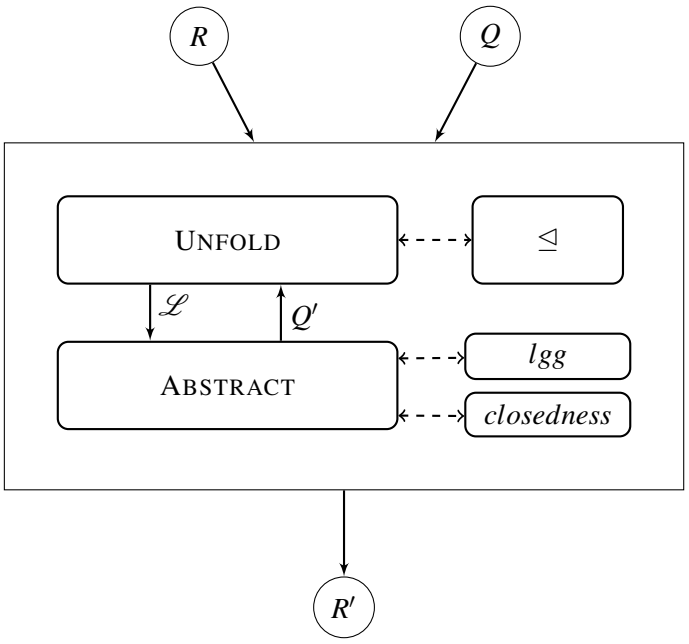


FIGURE 1.4: Narrowing-driven Partial Evaluation

1.2 Partial Evaluation of Maude Equational Theories

Functional rule-based languages that are encoded with equational reasoning capabilities offer a high-level approach to programming and analyzing complex software systems. However, partial evaluation has never been investigated in the context of expressive rule-based languages such as Maude, CafeOBJ, OBJ, ASF+SDF, and ELAN, which support:

1. rich type structures with sorts, subsorts and overloading; and
2. equational rewriting modulo combinations of associativity, commutativity, and/or identity axioms.

When we consider such advanced features, a more general framework extending NPE is needed. This thesis extends the NPE scheme of [Alpuente et al. 1998a] to deal with Maude programs (or more specifically, with Maude functional modules defining order-sorted equational theories). This means that, the key NPE components of [Alpuente et al. 1998a] have to be properly generalized to corresponding (order–sorted) *equational* notions (*modulo* axioms):

- *order-sorted equational unfolding*,
- *order-sorted equational closedness*,
- *order-sorted equational embedding*, and
- *order-sorted equational abstraction*.

As a result, the associated partial evaluation techniques become more sophisticated and powerful, but also more complex. Note also that the previous existing NPE framework is an instance of the equational NPE framework developed in this thesis.

In the remaining of this section, we briefly describe the different key NPE components of [Alpuente et al. 1998a] that have been generalized in this PhD thesis to corresponding (order–sorted) *equational* notions (*modulo* axioms) as they are the core of the partial evaluator Victoria [Victoria Website] that implements all the techniques developed in this thesis.

This section is organized as follows. In Section 1.2.1, we briefly summarize the main features of the Maude language. In Section 1.2.2, we recall the folding variant narrowing strategy that is used in Maude to narrow terms in convergent equational theories, together with an example that illustrates how it works. Section 1.2.3 presents the novel notion of equational closedness and how it is used within the specialization process. In Section 1.2.4, we describe the stopping criteria that are used to guarantee that the specialization process terminates. Finally, Section 1.2.5 describes a convenient post-processing renaming transformation that achieves further specialization by getting rid of redundant function symbols in the specialized program.

1.2.1 Equational Theories in Maude

Maude[1] is a high-level and high-performance language that implements *Rewriting Logic (RWL)* [Meseguer 1992] a very general logical and semantic framework in which different models of concurrent systems, distributed algorithms, programming languages, and software and hardware modeling languages can be naturally represented, executed, and analyzed as rewrite theories [Meseguer 2012]. Recently, an experimental open platform has been developed in [Garavel et al. 2018] that allows the performance of functional and algebraic programming languages to be compared, including CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML, Stratego/XT, and Tom (see references in [Garavel et al. 2018]). In the top 5 of the more efficient tools, Maude ranks second after Haskell. This is remarkable for at least two reasons: (i) Maude is not a compiled language but runs under an interpreter; (ii) Maude has quite sophisticated features (subtype polymorphism, pattern matching modulo associativity, commutativity and identity, reflection, strategies, objects, etc.) that have no equivalent in Haskell or other functional languages.

Roughly speaking, a *rewriting logic theory* seamlessly combines a term *rewriting system (TRS)* with an *equational theory* that may include equations and axioms so that rewrite steps are performed *modulo* the equations and axioms. In this thesis, we only consider Maude's equational theories and avoid any presentation of rewriting logic in general. An order-sorted equational theory is a triple $(\Sigma, B, \vec{E})$ where Σ is a typed signature of functions symbols, B is a set of commonly occurring algebraic axioms such as associativity, commutativity, and unity, and $\vec{E}$ is a collection of (possible conditional) Σ-equations and membership axioms (i.e., axioms that assert the type or *sort* of some terms). In addition, to rewriting in an equational theory $(\Sigma, B, \vec{E})$, an order-sorted equational theory $(\Sigma, B, \vec{E})$ can also be symbolically executed in Maude by performing narrowing with the oriented equations $\vec{E}$ modulo the axioms B [Clavel et al. 2016]. This form of narrowing is useful for equational unification and variant computation [Escobar et al. 2012], as well as for partial evaluation [Alpuente et al. 2018]. The *folding variant narrowing strategy* of [Escobar et al. 2012] is used for the case of narrowing with $\vec{E}$ modulo B in Maude. The main idea is to *fold*, by subsumption modulo B, the narrowing tree for $\vec{E}, B$, which can in

[1] The Maude system and full documentation are available at `http://maude.cs.uiuc.edu`.

practice result in a finite narrowing graph that symbolically and concisely summarizes the (generally infinite) $(\vec{E}, B)$ narrowing tree. When a convergent theory $(\Sigma, B, \vec{E})$ additionally satisfies the *finite variant property* [Comon-Lundh and Delaune 2005; Escobar et al. 2012], there exists a finite complete set of most general $(\vec{E},B)$-variants for each term t in the theory, where each $(\vec{E},B)$-variant of t consists of a substitution σ and the $(\vec{E},B)$-irreducible form of $t\sigma$. This notion is exemplified in the following section.

1.2.2 Variant generation

The *unfolding* technique for the specialization of equational theories, called *functional modules* in Maude, on which this thesis is based, is the folding variant narrowing strategy of [Escobar et al. 2012], which is able to memorize previous narrowing steps so that it avoids to reproduce useless or unnecessary narrowing steps. That is, folding variant narrowing allows the deployed folding variant narrowing tree to be seen as a graph, where certain "repeated" leaves are connected to other nodes by implicit "fold" arrows [Escobar et al. 2012].

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 ⟨*ModId*⟩ is the module where the command is executed [Clavel et al. 2016].

Example 1.3. *Consider the following equational theory [Clavel et al. 2016] (written in Maude syntax) for the "exclusive-or" symbol* `_*_` *that defines an exclusive union of sets of natural numbers,* `NatSet`*, such that* `X1 * X2` *is the set of natural numbers appearing in* `X1` *or* `X2`*, but not both, and* `mt` *represents the (empty set) identity element.*

```
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 Y Z : NatSet .
  eq [E1] : X * X     = mt [variant] .
  eq [E2] : X * mt    = X  [variant] .
  eq [E3] : X * X * Z = Z  [variant] .
endfm
```

The corresponding variant narrowing tree for the goal `X * X` *is shown in Figure 1.5. This theory has the finite variant property, i.e., the set of variants for each term is finite.*

In Maude, we can generate the variants for the given goal term as follows.

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

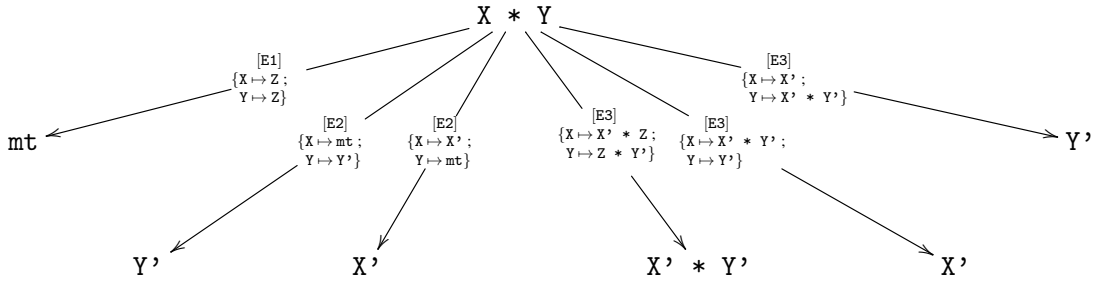


FIGURE 1.5: Folding variant narrowing tree for the goal `X * Y`.

Note that Maude delivers seven variants instead of the six ones depicted in Figure 1.5. This is because Variant 1 corresponds to the normalized goal term (modulo variable renaming). Also, note that Maude introduces fresh variables of two classes: #n:Sort (variables that are generated by the built-in unification modulo axioms algorithm) and %n:Sort (variables that are generated by variant-based unification or variant generation) [Clavel et al. 2016].

1.2.3 Order-sorted Equational closedness

Roughly speaking, the process of partially evaluating the equational theory $(\Sigma, B, \vec{E})$ w.r.t. a term t consists of: (i) deploying a finite $(\vec{E}, B)$-narrowing-tree for t by using the folding variant narrowing strategy, and (ii) extracting the specialized rules $t\sigma \Rightarrow r$ (resultants) for each narrowing derivation $t \leadsto^{+}_{\sigma,\vec{E},B} r$ in the tree. The *closedness* condition ensures that all calls that might occur during the specialization process are covered by the specialized program. Therefore, the closedness condition is critical for ensuring the completeness of the transformation. Informally, a term t is considered equationally closed w.r.t. a set of calls Q, iff it is an equational instance of a term of Q and the terms in the matching substitution are recursively E-closed by Q. For instance, consider an associativity function symbol & and the term $t = 0\&X\&Z$. We can say that t is closed w.r.t. $Q_1 = \{X\&Y\}$. However, it is not closed w.r.t. $Q_2 = \{X\&X\}$. In this thesis, equational closedness has been formally defined and very efficiently implemented in Maude as a part of our partial evaluator Victoria [Victoria Website].

1.2.4 Termination of the Equational Partial Evaluation process

A partial evaluator must ensure that the specialization process ends, so it has to deal with the *local* and *global* termination problems mentioned in Section 1.1.1 for the original NPE approach. In the following, we briefly describe the termination criteria applied in our partial evaluation framework.

Local Termination

The main idea of the local termination control is to ensure that, by observing some stop criterion, no narrowing tree is infinitely unfolded. The local termination strategy on which our partial evaluation framework is based on is the *equational homeomorphic embedding* relation. Homeomorphic embedding is a structural preorder under which a term t is greater than (i.e., it embeds) another term t', written as $t \triangleright t'$, if t' can be obtained from t by deleting some parts. The usefulness of embedding for ensuring termination is given by the following well-known

property of well-quasi-orderings: given a finite signature, for every infinite sequence of terms $t_1, t_2, \cdots$, there exist $i < j$ such that $t_i \triangleright t_j$. Therefore, to guarantee that the iterative computation of a sequence $t_1, t_2, \ldots, t_n$, the embedding can be used as a whistle [Leuschel 1998a]. That is, if a new expression t_{n+1} has to be added to the sequence, we first check whether t_{n+1} embeds any of the expressions already in the sequence. If that is the case, we say that $\triangleright$ whistles, i.e., it has detected (potential) non-termination and the computation must stop. Otherwise, t_{n+1} can be safely added to the sequence and the computation proceeds. For instance, if we have a sequence where the term $u = suc(0 + suc(X)) * suc(X + Y)$ is found after $v = suc(Y) * suc(X + 0)$, we must stop since u embeds v modulo the commutativity of $+$ and $*$.

In this thesis, we have defined and implemented two extensions of the homeomorphic embedding ("syntactically simpler") relation on nonground terms given in [Leuschel 1998a] to the order-sorted, *modulo axioms*, case of equational theories:

1. In [Alpuente et al. 2017a], we defined a naïve order-sorted equational extension of the homeomorphic embedding relation $\trianglelefteq_B$. However, this formulation did not scale to realistic problems. Furthermore, it did not consider types so that an embedding test such as `X:Bool` $\trianglelefteq_B$ `0 + suc(N:Nat)` succeeds.

2. In order to improve the preliminary proposal of [Alpuente et al. 2017a], we defined a more efficient formulation of order-sorted homeomorphic embedding modulo axioms $\check{\trianglelefteq}_B^{kosml}$ that may contain sorts, subsort polymorphism, and overloading in [Alpuente et al. 2018].

The formulation $\check{\trianglelefteq}_B^{kosml}$ given in [Alpuente et al. 2018] runs up to 6 orders of magnitude faster than the original definition of the homeomorphic embedding modulo equational axioms $\trianglelefteq_B$ in [Alpuente et al. 2017a]. This improvement in performance is achieved by taking advantage of Maude's powerful capabilities such as the efficiency of deterministic computations with equations versus non-deterministic computations with rewriting rules, or the use of non-strict definitions of the Boolean operators versus more speculative standard Boolean definitions [Clavel et al. 2007].

Global Termination

For global termination, NPE relies on an *abstraction operation* ensuring that the iterative construction of a sequence of partial narrowing trees terminates while still guaranteeing that the desired amount of specialization is retained and that the equational closedness condition is reached. Consider the set $\mathscr{L}$ of non-closed terms in the leaves of the unfolded narrowing trees (that have been stopped by applying the embedding test), and the current set Q of specialized calls (for which the unfolded narrowing trees have been generated). In order to avoid the construction of an infinite number of narrowing trees, instead of just taking the union of the sets $\mathscr{L}$ and Q, both sets (Q and $\mathscr{L}$) are *generalized*. Hence, the abstraction operation returns a safe generalization A of $Q \cup \mathscr{L}$ so that each expression in the set $Q \cup \mathscr{L}$ is closed w.r.t. A. For the generalization of terms, the new, equational abstraction operator is based on *order-sorted equational least general generalization (lgg_B)* [Alpuente et al. 2014b]. Roughly speaking, the generalization problem (also known as *anti-unification*) for two or more expressions means finding their least general generalization, i.e., the least general expression t such that all of the expressions are instances of t under appropriate substitutions. For instance, the expression `father(X,Y)` is a generalizer of both `father(john,sam)` and `father(tom,sam)`, but their least general generalizer (lgg), also known as the *most specific generalizer (msg)* and the *least common anti-instance*

(lcai), is `father(X,sam)`. Unlike the syntactical, untyped case, there is in general no unique least general generalization modulo axioms B, lgg_B, but a finite, minimal and complete set of lgg_B's for any two terms, so that any other generalizer has at least one of them as a B-instance [Alpuente et al. 2014b]. Formally, given an order-sorted signature Σ and a set of algebraic axioms B, a *generalization* modulo B of the nonempty set of Σ-terms $\{t_1,\ldots,t_n\}$ is a pair $\langle t,\Theta\rangle$, where $\Theta=\{\theta_1,\ldots,\theta_n\}$ is a set of substitutions, such that, for all $i=1,\ldots,n$, $t\theta_i =_B t_i$. The pair $\langle t,\Theta\rangle$ is the *least general generalization* modulo B of a set S of terms, written $lgg_B(S)$, if (1) $\langle t,\Theta\rangle$ is a generalization of S and (2) for every other generalization $\langle t',\Theta'\rangle$ of S, t' is more general than t modulo B.

In this thesis, we develop and implement [Alpuente et al. 2019a], a high-performance order-sorted least general generalization modulo B algorithm that runs up to 5 orders of magnitude faster than [Alpuente et al. 2014b].

1.2.5 Post-processing renaming modulo axioms

So far, by extending the NPE approach, we have been able to ensure the delivery of a resulting program whose equations only contain, in their right hand sides, expressions that are closed w.r.t. the final set of specialized calls. However, some redundant and unnecessary information might exist in the resulting program (e.g., redundant function symbols and unnecessary repetition of variables), so a post-partial evaluation process is necessary to eliminate such information and thus obtain a more efficient and readable program that can even get rid on the costly ACU-matching operations. Roughly speaking, the renaming process essentially introduces new function symbols for each specialized term and then replaces each call in the resulting program by a call to the corresponding renamed function.

Example 1.4. *Consider the following independent renaming for the specialized calls of Figure 1.3:* $\{X+suc(suc(0))\mapsto add_2(X)\}$ *and* $\{X+0\mapsto add_0(X)\}$.

The post-processing renaming derives the final renamed programs:

$add_2(0) = suc(suc(0))$ $add_2(suc(X')) = suc(add_2(X'))$	$add_0(0) = 0$ $add_0(suc(X')) = suc(add_0(X'))$
(a)	*(b)*

1.3 The Partial Evaluator Victoria

Victoria is the first automatic narrowing-driven partial evaluator for order-sorted equational theories written in Maude. Victoria is publicly available from [Victoria Website] and efficiently implements the different extensions developed in this thesis of the components of the equational NPE procedure of [Alpuente et al. 1998a] that are provided in this thesis. Figure 1.6 below shows the architecture of Victoria that corresponds to a classical web application. This consists of two main components, the *front-end* and the *back-end*, which are connected via a JSP-based layer that is implemented in Java (500 lines of Java source code). The front-end (or presentation layer) consists of 2.5K lines of JavaScript, HTML5, and CSS source code, and provides Victoria with an intuitive Web user interface. The back-end (or core engine) supports Victoria services and consists of about ten thousand lines of Maude code.

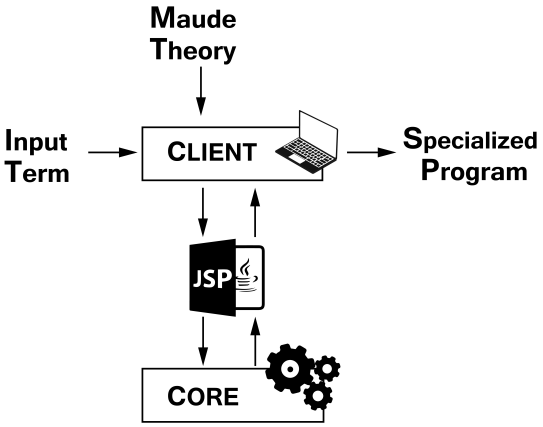


FIGURE 1.6: Architecture of Victoria.

Victoria is provided with an input panel, where we can enter the Maude program (or select a predefined Maude program example from the drop-down list) and the goal term, and then select the type of output program from the drop-down list (before or after renaming), before pressing the button "Evaluate ≫". Figures 1.7 and 1.8, respectively, show the input and output panels of Victoria, where we have the evaluated program and the independent renaming applied in Figure 1.8.

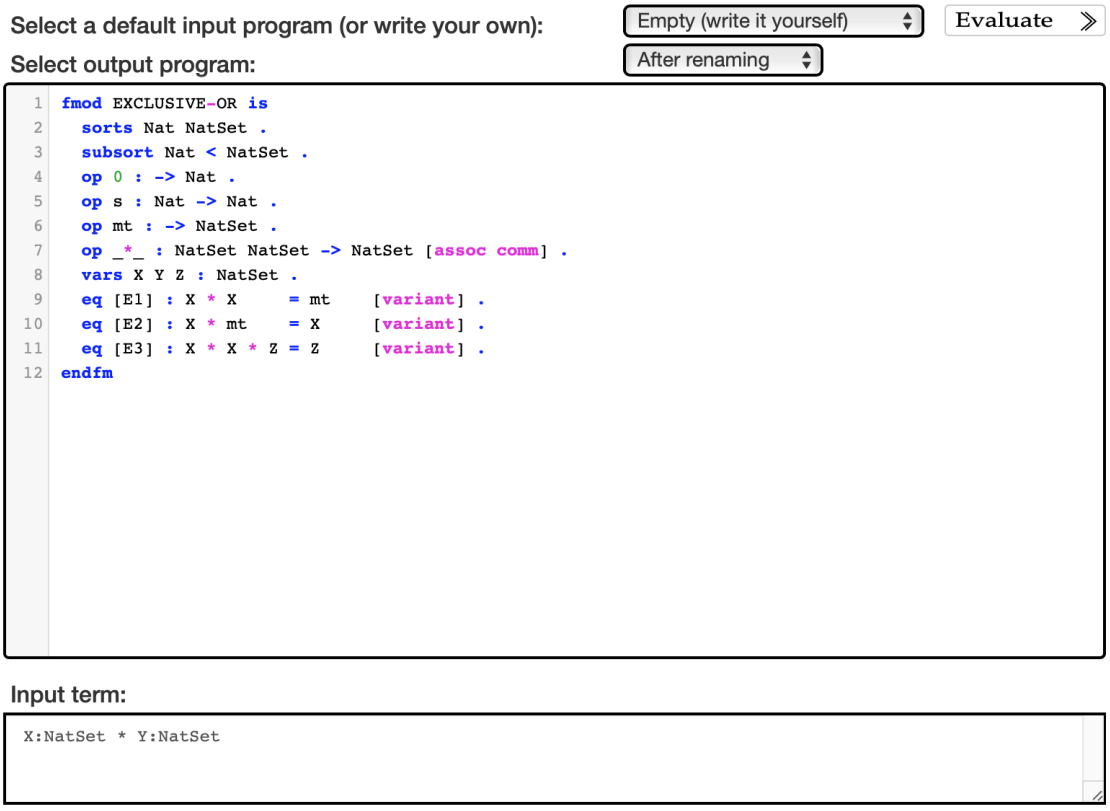


FIGURE 1.7: Input panel of Victoria.

The source code of the Victoria core, together with some benchmark problems, is available at [Victoria Website]. It can be executed in a localhost terminal console on the most recent version of Maude (version 2.7.1).

Evaluated program

```
fmod EXCLUSIVE-OR-EP is
 including BOOL .
 sort Nat .
 sort NatSet .
 subsort Nat < NatSet .
 op 0 :  -> Nat .
 op _*_ : NatSet NatSet -> NatSet [assoc comm]  .
 op f1 : NatSet NatSet -> NatSet .
 op mt :  -> NatSet .
 op s : Nat -> Nat .
 eq [eq1-1] : f1(X11%1:NatSet, X11%1:NatSet) = mt .
 eq [eq1-6] : f1(X11%1:NatSet, f1(X11%1:NatSet, X11%2:NatSet)) = X11%2:NatSet .
 eq [eq1-5] : f1(X11%1:NatSet, f1(X11%2:NatSet, X11%2:NatSet)) = X11%1:NatSet .
 eq [eq1-4] : f1(X11%1:NatSet, f1(X11%3:NatSet, X11%2:NatSet * X11%2:NatSet)) =
                f1(X11%1:NatSet, X11%3:NatSet) .
 eq [eq1-2] : f1(mt, X11%1:NatSet) = X11%1:NatSet .
 eq [eq1-3] : f1(mt, X11%1:NatSet) = X11%1:NatSet .
 endfm
```

Independent renaming

```
[*] X:NatSet * Y:NatSet ==> f1(X:NatSet, Y:NatSet)}
```

FIGURE 1.8: Output panel of Victoria.

1.4 Equational NPE in practice

The main motivation for building a partial evaluator is the efficiency that can be gained in the resulting specialized programs. This is why a partial evaluator is required to run fast and aims to produce residual programs that are semantically equivalent to the original program, but hopefully faster.

Let us show the power of our specialization technique through the following example that is borrowed from [Alpuente et al. 2017a], which illustrates the specialization of a program parser (w.r.t. a given grammar) into a very specialized parser that is similar to the approach proposed in [Jones et al. 1993].

Example 1.5. *Consider the following definition of a generic parser for languages that are generated by simple, right regular grammars (written in Maude[2] syntax). The* `PARSER` *module defines a constructor symbol* `_|_|_` *that represents the parser configurations, where the underscores correspond to its arguments. The first one represents the (terminal or non-terminal) symbol being processed. The second one represents the current string pending recognition. And finally, the third one stands for the considered grammar. It provides two non-terminal symbols,* `init` *and* `S`*, and three terminal symbols,* 0, 1*, and the finalizing mark* `eps` *(for* ε*, the empty string). These have been chosen to keep things simple, however they can be easily generalized to any terminal and non-terminal symbol by defining a Maude parameterized theory. Parsing a string* `(init |`

[2] In Maude 2.7.1, only equations with the attribute `variant` are used by the folding variant narrowing strategy.

`st | Γ`) *consists of using the rules of the grammar (in the opposite direction) to incrementally transform* `st` *until the final configuration* (`eps | eps | Γ`) *is reached.*

```
fmod PARSER is
  sorts Symbol NSymbol TSymbol String Production Grammar Parsing .
  subsort Production < Grammar .
  subsort TSymbol < String .
  subsorts TSymbol NSymbol < Symbol .
  ops 0 1 eps : -> TSymbol .
  ops init S : -> NSymbol .
  op mt : -> Grammar .
  op __ : TSymbol String -> String [right id: eps].
  op _->_ : NSymbol TSymbol -> Production .
  op _->_._ : NSymbol TSymbol NSymbol -> Production .
  op _;_ : Grammar Grammar -> Grammar [assoc comm id: mt] .
  op _|_|_ : Symbol String Grammar -> Parsing .
  var E : TSymbol .  var L : String .  var G : Grammar . vars N M : NSymbol .
  eq [end] :      N   | eps | (N -> eps) ; G =
                  eps | eps | (N -> eps) ; G [variant] .
  eq [continue] : N | E L | (N -> E . M) ; G =
                  M |  L  | (N -> E . M) ; G [variant] .
endfm
```

The second equation (labelled `[continue]`*) defines the general case of the parser, which given the configuration* (`N | E L | Γ`) *where* (`E L`) *is the string to be recognized, searches for the grammar production* (`N -> E . M`) *in* Γ *to recognize symbol* `E` *and then proceeds to recognize* `L` *starting from the non-terminal symbol* `M`*. We assume that the theory is convergent modulo axioms, i.e., there are no two grammar productions in* Γ *of the form* (`N -> E.`$\mathtt{M}_1$) *and* (`N -> E.`$\mathtt{M}_2$)*, which implies that there is always only one possible application of the equation* `[continue]`*. Also, note that the combination of subtypes and equational (algebraic) axioms allows for a very compact definition.*

For example, given the following grammar Γ *with five rules that generates the language* $\mathtt{0}^*\mathtt{1}^*$*:*

```
init -> eps ; init -> 0 . init ; init -> 1 . S ; S -> eps ; S -> 1 . S
```

the initial configuration (`init | 0 0 1 1 eps | Γ`) *is simplified into* (`init | 0 0 1 1 | Γ`) *by using right identity and is then deterministically rewritten as follows*

```
(init|0 0 1 1|Γ) →
(init|  0 1 1|Γ) →
(init|    1 1|Γ) →
( S  |      1|Γ) →
( S  |  eps  |Γ) →
( eps|  eps  |Γ)
```

The parser is encoded as a Maude equational theory that contains several language features that are unknown territory *for state-of-the-art (narrowing-driven) standard partial evaluation:*

(i) a hierarchy of sorts that defines the sets `TSymbol` *(terminal symbols) and* `NSymbol` *(non-terminal symbols) as two subsorts of the set* `Symbol` *of all grammar symbols; and*

(ii) a symbol `_;_` *that obeys the associativity (A), commutativity (C), and identity (U) axioms for representing grammars (meaning that they are handled as a multiset of productions), together with the symbol* `__` *that is used to represent the input string and obeys right identity.*

We can specialize the parsing program to the productions of a given grammar. For instance, consider the given grammar Γ *shown above and the input term* `(init | L | Γ)`*, where* `L` *is a logical variable of sort* `String`*. By using our partial evaluator* Victoria*, we construct the corresponding folding variant narrowing tree that is shown in Figure 1.9. The leaves* `(eps | eps | Γ)` *of the tree are constructor terms and cannot be unfolded, and the leaf* `(init | L' | Γ)` *is closed modulo ACU w.r.t. the root of the tree. However, the leaf* `(S | L'' | Γ)` *is not closed w.r.t. the root of the tree and, therefore, we need to apply the abstraction operation to it and obtain a new set of terms that are subsequently taken as the roots of the new narrowing trees.*

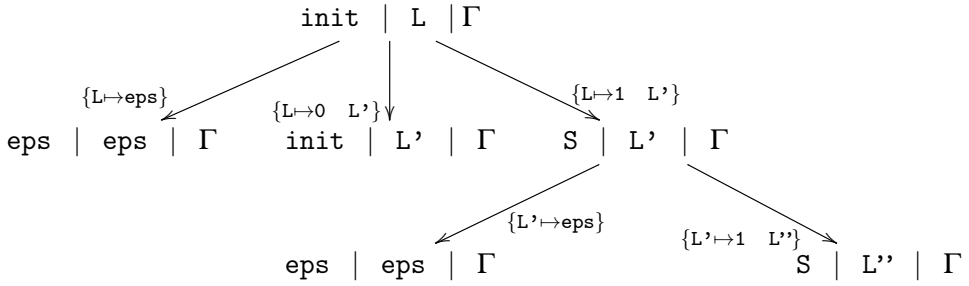


FIGURE 1.9: Folding variant narrowing tree for the goal `init | L |` Γ.

After applying the abstraction operation, the tree resulting for the new call `(S | L'' | Γ)` *is shown in Figure 1.10. Now all leaves in the trees of Figures 1.9 and 1.10 are closed w.r.t. their corresponding roots.*

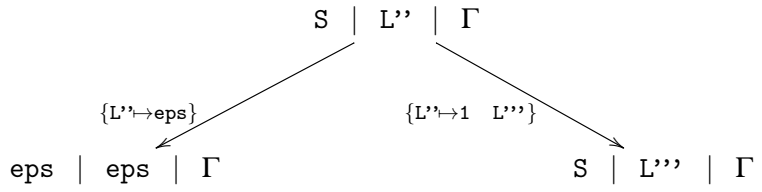


FIGURE 1.10: Folding variant narrowing tree for the goal `S | L'' |` Γ.

Therefore, Victoria *successfully returns the following specialized parsing equations:*

```
eq [1] :  init | eps   | Γ  =  eps  | eps | Γ [variant] .
eq [2] :  init | 0 L   | Γ  =  init | L   | Γ [variant] .
eq [3] :  init | 1     | Γ  =  eps  | eps | Γ [variant] .
eq [4] :  init | 1 1 L | Γ  =  S    | L   | Γ [variant] .
eq [5] :  S    | eps   | Γ  =  eps  | eps | Γ [variant] .
eq [6] :  S    | 1 L   | Γ  =  S    | L   | Γ [variant] .
```

```
eq [1] :  init || eps     =   eps  || eps [variant] .
eq [2] :  init || 0 L     =   init || L   [variant] .
eq [3] :  init || 1       =   eps  || eps [variant] .
eq [4] :  init || 1 1 L   =   S    || L   [variant] .
eq [5] :  S    || eps     =   eps  || eps [variant] .
eq [6] :  S    || 1 L     =   S    || L   [variant] .
```

where the third and fourth equations are specialized versions of the following equation

```
eq init | 1 L | Γ = S | L | Γ [variant] .
```

This is because the embedding test does not whistle in Figure 1.9 until the expression `S | L' | Γ` *is reached twice.*

By applying the post-processing renaming, we get a more readable specialized program that is still able to recognize the string `st` *by rewriting the simpler configuration* `(init | st | Γ)` *to the final configuration* `(eps | eps | Γ)`*. For instance, note that the third argument of the constructor symbol* `_|_|_` *in the left and right hand sides of the equations corresponds to the grammar* Γ*, and it is no longer necessary for the specialized program. By applying the indepent renaming* {`init | L | Γ` ↦ `finit(L)`, `S | L | Γ` ↦ `fS(L)`, `eps | eps | Γ` ↦ `feps`}*, we finally get the following specialized parsing equations:*

```
eq [1] : finit(eps)   = feps     [variant] .
eq [2] : finit(0 L)   = finit(L) [variant] .
eq [3] : finit(1)     = feps     [variant] .
eq [4] : finit(1 1 L) = fS(L)    [variant] .
eq [5] : fS(eps)      = feps     [variant] .
eq [6] : fS(1 L)      = fS(L)    [variant] .
```

This residual specialized program achieves a significant improvement in execution time compared to the original program, both before and after the renaming process, but even more noticeable after renaming. For example, given an input data string of five million elements, the parsing time with the original parser program is 275.334 ms, while for the specialized programs are: 2.058 ms before renaming (corresponding to 99.25% of improvement) and 1.685 ms after renaming (corresponding to 99.39% of improvement). Moreover, the final program after renaming moves from a program with ACU and Ur operators to a specialized program without axioms [Alpuente et al. 2017a].

1.5 Contributions of the Thesis

The main objective of this thesis is to develop a *partial evaluation framework* for order-sorted equational theories, that supports subsorts, subsort polymorphism, convergent rules (equations), and equational axioms. The contributions of this thesis are summarized as follows:

1. We develop Victoria, the *first partial evaluator* for Maude equational theories, which is based on a suitably extended version of the general NPE procedure of [Alpuente et al.

1998a] and is not only parametric w.r.t. the *unfolding rule* used to construct finite computation trees, but also w.r.t. an *abstraction operator* that is used to guarantee that only finitely many expressions are evaluated. For deploying narrowing-trees, Victoria relies on the *folding variant narrowing strategy*, which is optimal for convergent equational theories and efficiently computes *most general variants* modulo algebraic axioms [Escobar et al. 2012].

2. Folding variant narrowing computations can be extremely involved and are simply presented in constructed text format by Maude, often being too heavy to be debugged or even understood. We develop a graphical tool for exploring (variant) narrowing computations in Maude called GLINTS that allows: (i) exploring variant computations, (ii) determining whether a given theory satisfies the finite variant property, (iii) automatic checking of node *embedding* and *closedness* modulo axioms, and (iii) querying and inspecting selected parts of the variant trees [Alpuente et al. 2017b].

3. For the local termination control, our partial evaluator Victoria is based on using the *equational homeomorphic embedding* relation of [Alpuente et al. 2018], which efficiently handles sorts, subsort polymorphism, and overloading, and greatly improves the original formulation at [Alpuente et al. 2017a]. We develop the equational homeomorphic embedding checker in Maude called HEMS that implements all equational homeomorphic embedding formulations presented in [Alpuente et al. 2019b].

4. For the generalization process, we rely on the *order-sorted equational least general generalization*, first investigated in [Alpuente et al. 2014b], which we efficiently implemented in ACUOS^2, a high-performance order-sorted least general generalization modulo B tool for Maude described in [Alpuente et al. 2019a].

1.6 Plan of the Thesis

This thesis is structured as follows:

1. In Chapter 2, we recall some preliminary definitions that are needed to understand this thesis.

2. In Chapter 3, we describe GLINTS, the graphical tool for exploring variant narrowing computations in Maude. This chapter summarizes [Alpuente et al. 2017b].

3. In Chapter 4, we formalize the *order-sorted homeomorphic embedding module axioms* for Maude equational theories. This chapter summarizes [Alpuente et al. 2019b] that extends the work presented in [Alpuente et al. 2018].

4. In Chapter 5, we present ACUOS^2, the order-sorted equational least general generalization system. This chapter summarizes [Alpuente et al. 2019a].

5. In Chapter 6, we formulate our partial evaluation framework and the partial evaluator Victoria. This chapter summarizes [Alpuente et al. 2019c] that extends the work presented in [Alpuente et al. 2017a].

6. Finally, we present our conclusions in Chapter 7 and discuss direction for future work. Note that we have grouped together here the conclusions to each chapter.

1.7 List of Publications

The publications derived from this thesis are:

1. M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. **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), Edinburgh, UK, September 6-8, 2016*, volume 10184 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.

2. M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Sapiña. **Inspecting maude variants with GLINTS**. *Theory and Practice of Logic Programming*, 17(5-6):689-707, 2017.

3. M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. **Homeomorphic Embedding modulo Combinations of Associativity and Commutativity Axioms**. In *Proc. of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2018), Frankfurt/Main, Germany, September 4-6, 2018,* volume 11408 of *Lecture Notes in Computer Science*, pages 38–55. Springer, 2018.

4. M. Alpuente, D. Ballis, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. **ACUOS2: A High-performance System for Modular ACU Generalization with Subtyping and Inheritance**. In *Proc. of the 19th European Conference on Logics in Artificial Intelligence (JELIA 2019), Rende, Italy, May 8-10, 2019,* volume 11468 of *Lecture Notes in Artificial Intelligence*, pages 171–181. Springer, 2019.

5. M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. **Order-sorted Homeomorphic Embedding modulo Combinations of Associativity and/or Commutativity Axioms**. *Fundamenta Informaticae Journal*, 2019. Accepted for publication.

6. M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. **A Partial Evaluation Framework for Order-sorted Equational Programs modulo Axioms**. *Journal of Logical and Algebraic Methods in Programming*, 2019. Accepted for publication.

1.8 Developed Tools

The following software tools have been implemented during the development of this thesis.

1. GLINTS, a graphical tool for exploring (variant) narrowing computations in Maude. Available at [GLINTS Website].

2. HEMS, an equational homeomorphic embedding checker for Maude. Available at [HEMS Website].

3. ACUOS^2, a high performance least general generalization system for Maude rewrite theories. Available at [ACUOS^2 Website].

4. Victoria, a partial evaluator tool for Maude order-sorted equational theories. Available at [Victoria Website].

Chapter 2

Preliminaries

In this chapter, we provide some technical concepts that are required for the proper comprehension of this thesis. More specifically, Section 2.1 recalls some key concepts of order-sorted, rewriting logic theories and term rewriting [Meseguer 1992]. Section 2.2 introduces the notion of *decomposition* of an equational theory $(\Sigma, \mathscr{E})$. In Section 2.3, we give an overview of narrowing modulo equations [Meseguer and Thati 2007]. *Term variants* are explained in Section 2.4. In Section 2.5, we describe the variant narrowing strategy. Finally, Section 2.6 is devoted to *the folding variant narrowing strategy* of [Escobar et al. 2012].

2.1 Rewriting Logic and Term Rewriting

In this section, we recall some key concepts of order-sorted rewriting logic theories. More details can be found in [Meseguer 1992].

Terms, sorts and positions

We consider an *order-sorted signature* $(\Sigma, \mathsf{S}, \leq)$ that consists of a poset of sorts $(\mathsf{S}, \leq)$ and an $\mathsf{S}^* \times \mathsf{S}$-indexed family of sets $\Sigma = \{\Sigma_{\mathsf{s}_1 \ldots \mathsf{s}_n, \mathsf{s}}\}_{(\mathsf{s}_1 \ldots \mathsf{s}_n, \mathsf{s}) \in \mathsf{S}^* \times \mathsf{S}}$ of function symbols. The poset $(\mathsf{S}, \leq)$ of sorts for Σ is partitioned into equivalence classes $C_1, \ldots, C_n$ (called *connected components*) by the equivalence relation $(\leq \cup \geq)^+$. We assume that Σ is *preregular*, so each term t has a least sort, denoted $ls(t)$ (see [Goguen and Meseguer 1992]). Σ is also assumed to be *kind-complete*, that is, for each sort $\mathsf{s} \in \mathsf{S}$, its connected component in the poset $(\mathsf{S}, \leq)$ has a top sort under $\leq$, denoted $[\mathsf{s}]$ and called the connected component's *kind*, and for each function symbol $f \in \Sigma_{\mathsf{s}_1 \ldots \mathsf{s}_n, \mathsf{s}}$, there is also an $f \in \Sigma_{[\mathsf{s}_1] \ldots [\mathsf{s}_n], [\mathsf{s}]}$. An order-sorted signature can always be extended to be kind-complete [Meseguer 2016]. Maude automatically checks preregularity and adds a new "kind" sort $[\mathsf{s}]$ at the top of the connected component of each sort $\mathsf{s} \in \mathsf{S}$ specified by the user and automatically lifts each operator to the kind level. For technical reasons, it is useful to assume that Σ has no ad-hoc overloading. However, this assumption entails no real loss of generality: any Σ can be transformed into a semantically equivalent signature with no ad-hoc overloading (by symbol renaming). Finally, Σ is also assumed to be *sensible*, in the sense that for any two typings $f : \mathsf{s}_1 \ldots \mathsf{s}_n \longrightarrow \mathsf{s}$ and $f : \mathsf{s}'_1 \ldots \mathsf{s}'_n \longrightarrow \mathsf{s}'$ of an n-ary function symbol f, if s_i and s'_i are in the same connected component of $(\mathsf{S}, \leq)$ for $1 \leq i \leq n$, then s and s' are also in

the same connected component; this provides the right notion of *unambiguous* signature at the order-sorted level.

We assume an S-sorted family $\mathscr{X} = \{\mathscr{X}_s\}_{s\in S}$ of disjoint variable sets. $\mathscr{T}_\Sigma(\mathscr{X})_s$ and $\mathscr{T}_{\Sigma,s}$ denote the sets of terms and ground terms of sort s, respectively. Note that $s < s'$ (s is a subsort of s') implies the set of terms of sort s are a subset of the set of terms of sort s', i.e., $\mathscr{T}_\Sigma(\mathscr{X})_s \subseteq \mathscr{T}_\Sigma(\mathscr{X})_{s'}$. We also write $\mathscr{T}_\Sigma(\mathscr{X})$ and $\mathscr{T}_\Sigma$ for the corresponding term algebras, i.e., $\mathscr{T}_\Sigma(\mathscr{X}) = \cup_{s\in S}\mathscr{T}_\Sigma(\mathscr{X})_s$ and $\mathscr{T}_\Sigma = \cup_{s\in S}\mathscr{T}_{\Sigma,s}$. Throughout this thesis we assume that $\mathscr{T}_{\Sigma,s} \neq \emptyset$ for every sort s because this affords a simpler deduction system. The set of variables occurring in a term t is denoted by $\mathscr{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise. Let $\rightarrow \subseteq A \times A$ be a binary relation on a set A. We denote its transitive closure by $\rightarrow^+$, and its reflexive and transitive closure by $\rightarrow^*$. A sequence of syntactic objects $o_1, \ldots, o_n$ is denoted by $\bar{o}_n$.

A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). Positions are ordered by the *prefix* ordering: $p \leq q$, if $\exists w$ such that $p.w = q$. Given a term t, we let $\mathscr{P}os(t)$ and $\mathscr{P}os_\Sigma(t)$ respectively denote the set of positions and the set of non-variable positions of t (i.e., positions where a variable does not occur). The expression $t|_p$ denotes the *subterm* of t at position p, and $t[u]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term u.

Equational Theories and unification

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

A Σ-*equation* is an unoriented pair $t = t'$, where $t, t' \in \mathscr{T}_\Sigma(\mathscr{X})_s$ for some sort $s \in S$. Given Σ and a set $\mathscr{E}$ of Σ-equations, order-sorted equational logic induces a congruence relation $=_\mathscr{E}$ on $\mathscr{T}_\Sigma(\mathscr{X})$ (see [Meseguer 1997]). An *equational theory* $(\Sigma, \mathscr{E})$ is a pair with Σ an order-sorted signature and $\mathscr{E}$ a set of Σ-equations. We omit Σ when no confusion can arise.

A term t is more (or equally) general than t' modulo $\mathscr{E}$, denoted by $t \leq_\mathscr{E} t'$, if there is a substitution γ such that $t' =_\mathscr{E} t\gamma$. A substitution θ is more (or equally) general than σ modulo $\mathscr{E}$, denoted by $\theta \leq_\mathscr{E} \sigma$, if there is a substitution γ such that $\sigma =_\mathscr{E} \theta\gamma$, i.e., for all $x \in \mathscr{X}, x\sigma =_\mathscr{E} x\theta\gamma$. Also, $\theta \leq_\mathscr{E} \sigma\ [V]$ iff for all $x \in V, x\theta \leq_\mathscr{E} x\sigma$. We also define $t \simeq_\mathscr{E} t'$ iff $t \leq_\mathscr{E} t'$ and $t' \leq_\mathscr{E} t$; and similarly $\theta \simeq_\mathscr{E} \sigma$.

An $\mathscr{E}$-*unifier* for a Σ-equation $t = t'$ is a substitution σ such that $t\sigma =_\mathscr{E} t'\sigma$. $CSU_\mathscr{E}(t = t')$ denotes a *complete* set of unifiers for the equation $t = t'$ modulo $\mathscr{E}$. A set U of unifiers for the equation $t = t'$ is *complete* if, for any unifier γ of $t = t'$ there is a more general unifier σ in U. This means that all possible unifiers are subsumed or derivable from the ones in the complete set U. An $\mathscr{E}$-unification algorithm is *complete* if for any equation $t = t'$ it generates a complete set of $\mathscr{E}$-unifiers. Note that this set needs not be finite. A unification algorithm is said to be *finitary*

if it always terminates. Note that a complete and finitary $\mathscr{E}$-unification algorithm may not exist even if a complete and finite set of $\mathscr{E}$-unifiers exists.

Note that we use the words identities, equations and axioms to refer to algebraic properties although they have a different operational behaviour (see Baader et al. 2001). Note also that we often overload the equality symbol $=$ for equations in theories, for unification problems, and for syntactic equality (without any algebraic properties).

Rewrite Theories and Term Rewriting

A *rewrite theory* is a triple $\mathscr{R} = (\Sigma, \mathscr{E}, R)$, where $(\Sigma, \mathscr{E})$ is the equational theory modulo which we rewrite and R is a set of rewrite rules. Rules are of the form $l \to r$ where terms $l, r \in \mathscr{T}_\Sigma(\mathscr{X})_\mathsf{s}$ for some sort s are respectively called the *left-hand side* (or *lhs*) and the *right-hand side* (or *rhs*) of the rule and $\mathscr{V}ar(r) \subseteq \mathscr{V}ar(l)$. The set R of rules is required to be *sort-decreasing*, i.e., for each $l \to r$ in R, each $\mathsf{s} \in \mathsf{S}$, and each substitution σ, $r\sigma \in \mathscr{T}_\Sigma(\mathscr{X})_\mathsf{s}$ implies $l\sigma \in \mathscr{T}_\Sigma(\mathscr{X})_\mathsf{s}$.

We define the *one-step rewrite relation* on $\mathscr{T}_\Sigma(\mathscr{X})$ for the set of rules R as follows: $t \to_R t'$ if there is a position $p \in \mathscr{P}os(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$. The relation $\to_{R/\mathscr{E}}$ for rewriting modulo $\mathscr{E}$ is defined as $=_\mathscr{E} \circ \to_R \circ =_\mathscr{E}$. A term t is called *$R/\mathscr{E}$-irreducible* iff there is no term u such that $t \to_{R/\mathscr{E}} u$. A substitution σ is $R/\mathscr{E}$-irreducible if, for every $x \in \mathscr{X}$, $x\sigma$ is $R/\mathscr{E}$-irreducible. We say that the relation $\to_{R/\mathscr{E}}$ is *terminating* if there is no infinite sequence $t_1 \to_{R/\mathscr{E}} t_2 \to_{R/\mathscr{E}} \cdots t_n \to_{R/\mathscr{E}} t_{n+1} \cdots$. We say that the relation $\to_{R/\mathscr{E}}$ is *confluent* if whenever $t \to^*_{R/\mathscr{E}} t'$ and $t \to^*_{R/\mathscr{E}} t''$, there exists a term t''' such that $t' \to^*_{R/\mathscr{E}} t'''$ and $t'' \to^*_{R/\mathscr{E}} t'''$. A rewrite theory $(\Sigma, \mathscr{E}, R)$ is convergent if R is sort-decreasing and the relation $\to_{R/\mathscr{E}}$ is confluent and terminating. In a convergent order-sorted rewrite theory, for each term $t \in \mathscr{T}_\Sigma(\mathscr{X})$, there is a unique (up to $\mathscr{E}$-equivalence) $R/\mathscr{E}$-irreducible term t' that can be obtained by rewriting t to $R/\mathscr{E}$-irreducible or *normal* form, which is denoted by $t \to^!_{R/\mathscr{E}} t'$, or $t{\downarrow_{R/\mathscr{E}}}$ when t' is not relevant. For each $x \in Dom(\sigma)$, $\sigma{\downarrow_{R/\mathscr{E}}}$ is defined as $(\sigma{\downarrow_{R/\mathscr{E}}})(x) = x{\downarrow_{R/\mathscr{E}}}$. A substitution σ is $R/\mathscr{E}$-irreducible (normalized) iff $x\sigma$ is $R/\mathscr{E}$-irreducible for each $x \in Dom(\sigma)$. For a set of terms Q, we denote by $Q{\downarrow_{R/\mathscr{E}}}$ the set of normal forms of the elements in Q.

Since $\mathscr{E}$-congruence classes can be infinite, $\to_{R/\mathscr{E}}$-reducibility is undecidable in general. Therefore, $R/\mathscr{E}$-rewriting is usually implemented [Jouannaud et al. 1983] by $R,\mathscr{E}$-rewriting. We define the relation $\to_{R,\mathscr{E}}$ on $\mathscr{T}_\Sigma(\mathscr{X})$ by $t \to_{p,R,\mathscr{E}} t'$ (or simply $t \to_{R,\mathscr{E}} t'$) iff there is a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \to r$ in R, and a substitution σ such that $t|_p =_\mathscr{E} l\sigma$ and $t' = t[r\sigma]_p$. To ensure completeness of $R,\mathscr{E}$-rewriting w.r.t. $R/\mathscr{E}$-rewriting, we require *strict coherence*, ensuring that $=_\mathscr{E}$ is a bisimulation for $R,\mathscr{E}$-rewriting [Meseguer 2017]: for any Σ-terms u, u', v if $u =_\mathscr{E} u'$ and $u \to_{R,\mathscr{E}} v$, then there exists a term v' such that $u' \to_{R,\mathscr{E}} v'$ and $v =_\mathscr{E} v'$. Note that, assuming $\mathscr{E}$-matching is decidable, $\to_{R,\mathscr{E}}$ is decidable and notions such as confluence, termination, irreducible term, and normalized substitution are defined for $\to_{R,\mathscr{E}}$ straightforwardly [Meseguer 2017]. It is worth noting that Maude automatically provides $\mathscr{E}$-coherence completion for rules and equations.

2.2 Equational Theories as Rewrite Theories

Algebraic structures often involve axioms like associativity and/or commutativity of function symbols, which cannot be handled by ordinary term rewriting [Eker 2003] but instead are handled implicitly by working with congruence classes of terms. This is why an equational theory

is often decomposed into a disjoint union $\mathscr{E} = E \uplus B$, where B is a set of algebraic axioms (which are implicitly expressed in Maude as attributes of their corresponding operator using the `assoc`, `comm`, and `id` : keywords) that are used for B-matching, and E consists of (possibly conditional) equations that are implicitly oriented from left to right as a set $\vec{E}$ of rewrite rules (and operationally used as simplification rules modulo B). By doing this, a (well-behaved) rewrite theory $(\Sigma, B, \vec{E})$ is defined, with $\vec{E} = \{t \to t' \mid t = t' \in E\}$, which satisfies all of the conditions that we need. This is formalized by the notion of *decomposition* of the equational theory $(\Sigma, \mathscr{E})$ as follows.

Definition 2.1 (Decomposition [Escobar et al. 2009b])**.** Let $(\Sigma, \mathscr{E})$ be an order-sorted equational theory. We call $(\Sigma, B, \vec{E})$ a *decomposition* of $(\Sigma, \mathscr{E})$ if $\mathscr{E} = E \uplus B$ and $(\Sigma, B, \vec{E})$ is an order-sorted rewrite theory satisfying the following properties:

1. B is *regular*, i.e., for each $t = t'$ in B, we have $Var(t) = Var(t')$, and *linear*, i.e., for each $t = t'$ in B, each variable occurs only once in t and in t'.
2. B is *sort-preserving*, i.e., for each $t = t'$ in B and substitution σ, we have $t\sigma \in \mathscr{T}_\Sigma(\mathscr{X})_\mathsf{s}$ iff $t'\sigma \in \mathscr{T}_\Sigma(\mathscr{X})_\mathsf{s}$. Furthermore, for each equation $t = t'$ in B, all variables in $Var(t)$ and $Var(t')$ have a common top sort.
3. B has a finitary and complete unification algorithm, which implies that B-matching is decidable.
4. The rewrite rules in $\vec{E}$ are *convergent*, i.e., confluent, terminating, and strictly coherent modulo B, and *sort-decreasing*.

We often abuse notation and say that $(\Sigma, B, \vec{E})$ is a decomposition of an order-sorted equational theory $(\Sigma, \mathscr{E})$ even if $\mathscr{E} \neq E \uplus B$ but E is instead the explicitly extended B-coherent completion of a set E' such that $\mathscr{E} = E' \uplus B$.

Given the rewrite theory $(\Sigma, B, \vec{E})$, it is common to split the signature Σ into two disjoint sets: defined symbols and constructor symbols. *Defined symbols* are defined as $\mathscr{D}_{\vec{E}} = \{f \in \Sigma \mid \exists f(t_1, \ldots, t_n) \to r \in \vec{E}\}$, and *constructors* are defined as $\mathscr{C}_{\vec{E}} = \Sigma \setminus \mathscr{D}_{\vec{E}}$.

In the following, we often consider rewrite theories (Σ, B, R) that are a decomposition of an order-sorted equational theory.

2.3 Narrowing in Rewriting Logic

Narrowing generalizes term rewriting by allowing free variables in terms (as in logic programming) and by performing unification (at non-variable positions) instead of matching in order to (non–deterministically) reduce the term. Function definition and evaluation are thus embedded within a symbolic logical framework and features such as existentially quantified variables, unification, and function inversion become available.

Definition 2.2 ((R,B)-narrowing [Meseguer and Thati 2007])**.** Let $\mathscr{R} = (\Sigma, B, R)$ be an order-sorted rewrite theory. The (R,B)-*narrowing* relation on $\mathscr{T}_\Sigma(\mathscr{X})$ is defined as $t \leadsto_{\sigma,R,B} t'$ (or just $t \leadsto_\sigma t'$) if there exist $p \in \mathscr{P}os_\Sigma(t)$, a (renamed apart) rule $l \to r$ in R, and a B-unifier σ of $t|_p$ and l such that $t' = (t[r]_p)\sigma$. The narrowing step $t \leadsto_{\sigma,R,B} t'$ is also called a (R,B)-*narrowing step*. A term t is (R,B)-*narrowable* if there exist σ and t' such that $t \leadsto_{\sigma,R,B} t'$. Given the narrowing

sequence $\alpha : (t_0 \leadsto_{\theta_1} t_1 \cdots \leadsto_{\theta_n} t_n)$, the computed substitution of α is $\theta = (\theta_1 \ldots \theta_n)|_{Var(t_0)}$ and we may write $t_0 \leadsto^n_\theta t_n$.

Since (R,B)-narrowing has quite a large search space, suitable strategies are needed to improve the efficiency of narrowing by getting rid of useless computations.

First, we define the notion of a narrowing strategy. Given a (R,B)-narrowing sequence $\alpha : (t_0 \leadsto_{\theta_1} t_1 \cdots \leadsto_{\theta_n} t_n)$, we denote by α_i the narrowing sequence $\alpha_i : (t_0 \leadsto_{\theta_1} t_1 \cdots \leadsto_{\theta_i} t_i)$, which is a prefix of α. Given an order-sorted rewrite theory $\mathscr{R}$, we denote by $Full_{\mathscr{R}}(t)$ the (possibly infinite) set of all (R,B)-narrowing sequences stemming from t.

Definition 2.3 (Narrowing Strategy)**.** A *narrowing strategy* is a function of two arguments: a rewrite theory $\mathscr{R} = (\Sigma, B, R)$ and a term $t \in \mathscr{T}_\Sigma(\mathscr{X})$, which we denote by $\mathscr{S}_{\mathscr{R}}(t)$, such that $\mathscr{S}_{\mathscr{R}}(t) \subseteq Full_{\mathscr{R}}(t)$. We require $\mathscr{S}_{\mathscr{R}}(t)$ to be prefix closed, i.e., for each narrowing sequence $\alpha \in \mathscr{S}_{\mathscr{R}}(t)$ of length n, and each $i \in \{1,\ldots,n\}$, we also have $\alpha_i \in \mathscr{S}_{\mathscr{R}}(t)$.

Narrowing strategies for rewrite theories that are complete (i.e., for every solution, it computes a more general answer) under suitable conditions have been investigated in [Meseguer and Thati 2007; Thati and Meseguer 2006].

Example 2.1. *The equational theory for exclusive–or has a decomposition into $\vec{E}$ consisting of the (implicitly oriented) equations (3)–(5) below, and B the associativity and commutativity (AC) axioms for symbol $\oplus$:*

$$X \oplus 0 = X \quad (3) \qquad X \oplus X = 0 \quad (4) \qquad X \oplus X \oplus Y = Y \quad (5)$$

Note that equations (3)–(4) are not strictly AC-coherent, but adding equation (5) is sufficient to recover that property (see [Durán and Meseguer 2010; Viry 2002]).

Given the term $t = X \oplus Y$, the following $(\vec{E},B)$–narrowing steps can be proved (we only include the bindings for the variables in the input term)

$$\begin{array}{l}
X \oplus Y \leadsto_{\phi_1} X' \quad \textit{using } \phi_1 = \{X \mapsto 0, Y \mapsto X'\} \textit{ and Equation (3)} \\
X \oplus Y \leadsto_{\phi_2} X' \quad \textit{using } \phi_2 = \{X \mapsto X', Y \mapsto 0\} \textit{ and Equation (3)} \\
X \oplus Y \leadsto_{\phi_3} 0 \quad \textit{using } \phi_3 = \{X \mapsto X', Y \mapsto X'\} \textit{ and Equation (4)} \\
X \oplus Y \leadsto_{\phi_4} Y' \quad \textit{using } \phi_4 = \{X \mapsto Y' \oplus X', Y \mapsto X'\} \textit{ and Equation (5)} \\
X \oplus Y \leadsto_{\phi_5} Y' \quad \textit{using } \phi_5 = \{X \mapsto X', Y \mapsto Y' \oplus X'\} \textit{ and Equation (5)} \\
X \oplus Y \leadsto_{\phi_6} U \oplus Y' \quad \textit{using } \phi_6 = \{X \mapsto X' \oplus U, Y \mapsto X' \oplus Y'\} \textit{ and Equation (5)}
\end{array}$$

As explained above, in order to provide a finitary and complete unification algorithm for a decomposition $(\Sigma, B, \vec{E})$, two narrowing strategies are defined in [Escobar et al. 2012]: *variant narrowing* and *folding variant narrowing*. These notions are formalized in the following sections.

2.4 Term Variants

Intuitively, an $(\vec{E},B)$-*variant* of a term t is the $(\vec{E},B)$-irreducible form of an *instance* $t\sigma$ of t. That is, the variants of t are all of the possible $(\vec{E},B)$-irreducible terms to which instances of t evaluate. Note that variant terms are normalized.

Definition 2.4 (Term variant [Comon-Lundh and Delaune 2005])**.** Given a term t and an equational theory $(\Sigma, E \uplus B)$ with a decomposition $(\Sigma, B, \vec{E})$, we say that (t', θ) is a *variant* of t if $t' =_B (t\theta)\downarrow_{\vec{E},B}$, where $Dom(\theta) \subseteq Var(t)$ and $Ran(\theta) \cap Var(t) = \emptyset$.

Example 2.2. *Consider the following specification for addition of natural numbers:*

$$0 + M = M$$
$$s(N) + M = s(N + M)$$

The set of variants for the term $N + 0$ is infinite, since we have $(0, \{N \mapsto 0\})$, $(s(0), \{N \mapsto s(0)\})$, ..., $(s^k(0), \{N \mapsto s^k(0)\})$. Analogously, the variants of the term $0 + M$ are $(0, \{M \mapsto 0\})$, $(s(0), \{M \mapsto s(0)\})$, ..., $(s^k(0), \{M \mapsto s^k(0)\})$.

In order to capture when a newly generated variant is subsumed by a previously generated one, we introduce the notion of *variant preordering with normalization*.

Definition 2.5 (More General Variant [Escobar et al. 2012])**.** Given a decomposition $(\Sigma, B, \vec{E})$ and two term variants $(t_1, \theta_1), (t_2, \theta_2)$ of a term t, we write $(t_1, \theta_1) \leq_{\vec{E},B} (t_2, \theta_2)$, meaning (t_1, θ_1) is a more general variant of t than (t_2, θ_2), iff there is a substitution ρ such that $(\theta_1\rho)|_{Var(t)} =_B (\theta_2\downarrow_{\vec{E},B})|_{Var(t)}$ and $t_1\rho =_B t_2$.

Example 2.3. *The term $N + M$ has an infinite set of most general variants in the theory of Example 2.2, since we have $(M, \{N \mapsto 0\})$, $(s(M), \{N \mapsto s(0)\})$, ..., $(s^k(M), \{N \mapsto s^k(0)\})$. However, note that the variant $(0, \{N \mapsto 0, M \mapsto 0\})$ is subsumed by $(M, \{N \mapsto 0\})$ and is therefore discarded from the set of most general variants. The set of most general variants of the term $0 + M$ is finite and is $\{(M, \varepsilon)\}$.*

An equational theory has the *finite variant property* (FVP) iff there is a finite and complete set of most general variants for each term. We say an equational theory is a *finite variant theory* if it has the FVP. The specification of natural numbers of Example 2.2 is not a finite variant theory, since the term $N + M$ has an infinite number of most general variants, as shown in Example 2.3. The equational theory for exclusive-or of Example 2.1 is a finite variant theory as it is the following theory for Boolean expressions.

Example 2.4. *Consider the following theory that declares the two Boolean constants* `true` *and* `false`*. The key things 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 .
  op not : Bool -> Bool .
  ops _and_ _or_ : Bool Bool -> Bool [assoc comm] .
  vars X Y : Bool .
  eq not(true) = false   [variant] .
  eq not(false) = true   [variant] .
  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:* $(\texttt{X and Y}, id), (\texttt{Y}, \{\texttt{X} \mapsto \texttt{true}\}), (\texttt{X}, \{\texttt{Y} \mapsto \texttt{true}\}), (\texttt{false}, \{\texttt{X} \mapsto \texttt{false}\}), (\texttt{false}, \{\texttt{Y} \mapsto \texttt{false}\})$. *Similarly, there are five most general variants for "*`X or Y`*".*

It is generally undecidable whether an equational theory has the FVP [Bouchard et al. 2013]; a semi-decision procedure is given in [Cholewa et al. 2014; Meseguer 2018] that works well in practice, and another technique based on the dependency pair framework is given in [Escobar et al. 2012]. The procedure in [Cholewa et al. 2014] is implemented in [Alpuente et al. 2017b] and 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 [Cholewa et al. 2014].

2.5 The variant narrowing strategy

Given a decomposition $(\Sigma, B, \vec{E})$, applying narrowing without any restriction can be very wasteful due to two main sources: (i) for axioms B such as associativity-commutativity, the number of B-unifiers of an equation can be quite large; and (ii) if we narrow a term in all possible positions, the narrowing tree may grow in an explosive way. Let us first motivate the variant narrowing strategy with two ideas. First, for computing variants in a decomposition we are only interested in *normalized terms* and *normalized substitutions*, so we can restrict our interest to narrowing derivations that provide only normalized substitutions and end in normalized terms, whereas the unrestricted narrowing formalized in Definition 2.2 does not ensure that.

Example 2.5. *Continuing with Example 2.1, due to the prolific AC-unification algorithm there are some redundant narrowing steps with non-normalized substitutions, such as*

$$
\begin{array}{l}
X \oplus Y \leadsto_{\phi_7} X' \oplus U \quad \textit{using } \phi_7 = \{X \mapsto X' \oplus 0, Y \mapsto U\} \textit{ and Equation (3)} \\
X \oplus Y \leadsto_{\phi_8} U \oplus X' \quad \textit{using } \phi_8 = \{X \mapsto U, Y \mapsto 0 \oplus X'\} \textit{ and Equation (3)} \\
X \oplus Y \leadsto_{\phi_9} Y' \quad \textit{using } \phi_9 = \{X \mapsto X' \oplus X', Y \mapsto Y'\} \textit{ and Equation (5)} \\
X \oplus Y \leadsto_{\phi_{10}} Y' \quad \textit{using } \phi_{10} = \{X \mapsto Y', Y \mapsto X' \oplus X'\} \textit{ and Equation (5)} \\
X \oplus Y \leadsto_{\phi_{11}} Y' \oplus U \quad \textit{using } \phi_{11} = \{X \mapsto X' \oplus X' \oplus Y', Y \mapsto U\} \textit{ and Equation (5)} \\
X \oplus Y \leadsto_{\phi_{12}} U \oplus Y' \quad \textit{using } \phi_{12} = \{X \mapsto U, Y \mapsto X' \oplus X' \oplus Y'\} \textit{ and Equation (5)}
\end{array}
$$

For instance, note that the narrowing step with substitution ϕ_9 is not needed because the same effect is achieved with the normalized substitution ϕ_1. Indeed, the narrowing `search` *command of Maude [Clavel et al. 2009], which performs full (i.e., unrestricted) narrowing, (non–deterministically) computes* 124 *different narrowing steps from term $X \oplus Y$. When we consider narrowing sequences instead of single steps, we can easily get a combinatorial explosion, since we have another* 124 *different narrowing steps after any of the following ones: $X \oplus Y \leadsto_{\phi_6} Z_1 \oplus Z_2$, $X \oplus Y \leadsto_{\phi_8} Z_1 \oplus Z_2$, or $X \oplus Y \leadsto_{\phi_{11}} Z_1 \oplus Z_2$. Also, there are many infinite narrowing sequences, such as the one repeating substitution ϕ_6 again and again: $X \oplus Y \leadsto_{\phi_6} Z_1 \oplus Z_2 \leadsto_{\phi_6'} Z_1' \oplus Z_2' \leadsto_{\phi_6''} Z_1'' \oplus Z_2'' \leadsto \cdots$ where $\phi_6' = \{Z_1 \mapsto U' \oplus Z_1', Z_2 \mapsto U' \oplus Z_2'\}$ and $\phi_6'' = \{Z_1' \mapsto U'' \oplus Z_1'', Z_2' \mapsto U'' \oplus Z_2''\}$.*

Our second idea is to give priority to *most general* narrowing steps, instead of more instantiated ones, and to select one and only one narrowing step among those having the same generality, following a don't care approach. This has three implications. The first is that the

most general narrowing steps are rewriting steps (if any), and thus any (deterministic) rewrite step should be taken before exploring (possibly non-deterministic) narrowing steps. This resembles the optimization of narrowing known as *normalizing narrowing* (see, e.g., [Hanus 1994]). Thanks to convergence modulo B, as soon as a rewrite step $\rightarrow_{\vec{E},B}$ is enabled in a term that also has narrowing steps $\rightsquigarrow_{\vec{E},B}$, such a rewrite step is always taken before any further narrowing steps are applied. The idea of normalizing terms before any narrowing step is taken is consistent with the implementation of rewriting logic [Viry 2002], where deterministic rewrite steps (with equations) are given priority w.r.t. more expensive, non–deterministic rewrite steps (with rules). The second implication is that variant narrowing goes much further than just giving priority to rewrite steps by filtering out all narrowing steps that do not compute most general substitutions. Namely, given two narrowing steps $t \rightsquigarrow_{\sigma_1,\vec{E},B} t_1$ and $t \rightsquigarrow_{\sigma_2,\vec{E},B} t_2$ in a decomposition $(\Sigma, B, \vec{E})$ such that $\sigma_1 \leq_B \sigma_2$, we can safely disregard the narrowing step using σ_2 without losing completeness (c.f. [Escobar et al., 2012, Theorem 4]). The third implication is that we can pack together, in the same equivalence class, all narrowing steps with equally general substitutions and select just one of them as the class representative, thanks to convergence modulo B (see [Escobar et al. 2012] for further details).

Example 2.6. *There are nearly* 150 *unrestricted narrowing steps for the term* $X \oplus Y \oplus X \oplus Y$ *in the equational theory of Example 2.1 (recall that the subterm* $X \oplus Y$ *had* 124 *narrowing steps but the variant narrowing strategy defined below computes only the six narrowing steps of Example 2.1). For the term* $X \oplus Y \oplus X \oplus Y$*, variant narrowing recognizes that the term is not yet normalized, e.g.,* $X \oplus Y \oplus X \oplus Y \rightarrow 0$ *(by using Equation 4), and such a rewriting step is more general than any other narrowing step from* t*. Thus, such an exceptionally large number of narrowing steps can be disregarded by just choosing to rewrite the term. Note that there are two other rewrite steps* $X \oplus Y \oplus X \oplus Y \rightarrow Y \oplus Y$ *(by using Equation 4) and* $X \oplus Y \oplus X \oplus Y \rightarrow X \oplus X$ *(by using Equation 4), and equational rewriting in Maude will choose the one that rewrites the maximal* $\oplus$*-term possible due to implicit coherence extensions for rewriting (see [Durán and Meseguer 2010; Viry 2002]).*

On the other hand, one of the 124 unrestricted narrowing steps that are enabled for the more general term $X \oplus Y$ *is*

$$X \oplus Y \rightsquigarrow_{\mu} U_1 \oplus U_2 \oplus U_3 \oplus U_4 \quad \begin{array}{l} \text{using the B-unifier} \\ \mu = \{X \mapsto U \oplus U_1 \oplus U_2, Y \mapsto U \oplus U_3 \oplus U_4\} \\ \text{and Equation (5)} \end{array}$$

which is an AC-instance of the narrowing step using substitution ϕ_6 *shown above:*

$$X \oplus Y \rightsquigarrow_{\phi_6} Z_1 \oplus Z_2 \quad \text{using } \phi_6 = \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\} \text{ and Equation (5)}$$

Both narrowing steps are fired by Equation (5), but variant narrowing does discard the less general narrowing step with μ*, keeping only the more general narrowing step with* ϕ_6*.*

These optimizations are formalized as follows. First, a preorder between narrowing steps is introduced that defines when a narrowing step is more general than another narrowing step.

Definition 2.6 (Preorder and equivalence of narrowing steps [Escobar et al. 2012])**.** Given a decomposition $(\Sigma, B, \vec{E})$, consider two narrowing steps $\alpha_1 : t \rightsquigarrow_{\sigma_1,\vec{E},B} s_1$ and $\alpha_2 : t \rightsquigarrow_{\sigma_2,\vec{E},B} s_2$. Let $V = Var(t)$. We write $\alpha_1 \preceq_B \alpha_2$ if $\sigma_1 \leq_B \sigma_2[V]$ and $\alpha_1 \prec_B \alpha_2$ if $\sigma_1 <_B \sigma_2[V]$ (i.e., σ_1 is

strictly more general than σ_2 on V). We write $\alpha_1 \simeq_B \alpha_2$ if $\sigma_1 \simeq_B \sigma_2[V]$, i.e. $\alpha_1 \preceq_B \alpha_2$ and $\alpha_2 \preceq_B \alpha_1$.

The relation $\alpha_1 \simeq_B \alpha_2$ between narrowing steps defines a set of equivalence classes of narrowing steps. In what follows, we will be interested in choosing a unique representative $\underline{\alpha} \in [\alpha]_{\simeq_B}$ in each equivalence class of narrowing steps from t. Therefore, $\underline{\alpha}$ will always denote the chosen unique representative $\underline{\alpha} \in [\alpha]_{\simeq_B}$ that is minimal w.r.t. the order $\preceq_B$.

The relation $\preceq_B$ provides an improvement on narrowing executions in two ways. First, narrowing steps with more general computed substitutions will be selected instead of narrowing steps with more specific computed substitutions. As a particular case, when both a rewriting step and a narrowing step are available, the rewriting step will always be chosen. Second, the relation $\simeq_B$ provides a further optimization, since just one narrowing (or rewriting) step is chosen for each equivalence class, which further reduces the width of the narrowing tree.

The described strategy is formalized by the notion of *variant narrowing*.

Definition 2.7 (Variant Narrowing [Escobar et al. 2012])**.** Given a decomposition $(\Sigma, B, \vec{E})$ and a narrowing step $\alpha : t \leadsto_{\sigma,\vec{E},B} t'$, α is a *variant narrowing step* if it satisfies: (i) $\sigma|_{Var(t)}$ is $(\vec{E}, B)$-irreducible and (ii) $\underline{\alpha}$ is the chosen unique representative of its $\simeq_B$-equivalence class.

Following the notation of [Escobar et al. 2012], a variant narrowing step from t to t' in $(\Sigma, B, \vec{E})$ with substitution σ is denoted as $t \leadsto_{\sigma,\underline{\vec{E}},B} t'$.

2.6 The folding variant narrowing strategy

The variant narrowing strategy defined above is a strategy in the sense of Definition 2.3, i.e., it always returns a subset of the narrowing steps that are available for each term. Note, however, that it has no memory of previous steps –just the input term to be narrowed– hence, it incurs no memory overhead. More sophisticated strategies can be developed by introducing some sort of memory that can avoid the repeated generation of useless or unnecessary computation steps. This is the case of the folding narrowing strategy of [Escobar et al. 2012], which, when combined with the variant narrowing strategy, provides the *folding variant narrowing strategy* which is complete for variant generation of a term and it terminates when the input term has a finite set of *most general variants.*

In Definition 2.8 below, we introduce a *folding narrowing* relation on term variants. Folding narrowing allows the deployed variant narrowing tree to be seen as a graph, where some leaves are connected to other nodes by implicit "fold" arrows. This definition normalizes each computed variant, which is not performed in the original definition of [Escobar et al. 2012]. Note that we easily extend the variant narrowing strategy to variants, i.e., $(t, \theta) \leadsto_{\sigma,\underline{\vec{E}},B} (t', \theta')$ iff $t \leadsto_{\sigma,\underline{\vec{E}},B} t'$ and $\theta' = \theta\sigma$.

Definition 2.8 (Folding Variant Narrowing Strategy)**.** Let $\mathscr{R} = (\Sigma, B, \vec{E})$ be a decomposition. Given a Σ-term t, the frontier from $I = (t, id)$ is defined as

$$
\begin{aligned}
&Frontier(I)_0 = (t{\downarrow_{\vec{E},B}}, id),\\
&Frontier(I)_{n+1} = \{(y{\downarrow_{\vec{E},B}}, (\rho\sigma){\downarrow_{\vec{E},B}}) \mid (\exists (z,\rho) \in Frontier(I)_n : (z,\rho) \leadsto_{\sigma,\underline{\vec{E}},B} (y, \rho\sigma)) \wedge\\
&\qquad\qquad (\nexists k \leq n, (w,\tau) \in Frontier(I)_k : (w,\tau) \ \leq_{\vec{E},B} \ (y, \rho\sigma))\},\\
&\qquad\qquad n \geq 0
\end{aligned}
$$

The folding variant narrowing strategy, denoted by $VN^{\circlearrowleft}_{\mathscr{R}}$, is defined as

$$VN^{\circlearrowleft}_{\mathscr{R}}(t) = \{\alpha \mid \alpha : t \leadsto^{k}_{\sigma,\vec{E},B} t' \wedge \exists k \geq 0 : (t',\sigma) \in \mathit{Frontier}((t,id))_k\}$$

Example 2.7. *Using the term $X \oplus Y$, we get the following $VN^{\circlearrowleft}_{\mathscr{R}}$ steps in the equational theory of Example 2.1, where all substitutions are normalized.*

(i) $(X \oplus Y, id) \leadsto_{\phi_1} (Z, \phi_1)$, *using Equation (3) and substitution* $\phi_1 = \{X \mapsto 0, Y \mapsto Z\}$,

(ii) $(X \oplus Y, id) \leadsto_{\phi_2} (Z, \phi_2)$, *using Equation (3) and substitution* $\phi_2 = \{X \mapsto Z, Y \mapsto 0\}$,

(iii) $(X \oplus Y, id) \leadsto_{\phi_3} (Z, \phi_3)$, *using Equation (5) and substitution* $\phi_3 = \{X \mapsto Z \oplus U, Y \mapsto U\}$,

(iv) $(X \oplus Y, id) \leadsto_{\phi_4} (Z, \phi_4)$, *using Equation (5) and substitution* $\phi_4 = \{X \mapsto U, Y \mapsto Z \oplus U\}$,

(v) $(X \oplus Y, id) \leadsto_{\phi_5} (0, \phi_5)$, *using Equation (4) and substitution* $\phi_5 = \{X \mapsto U, Y \mapsto U\}$,

(vi) $(X \oplus Y, id) \leadsto_{\phi_6} (Z_1 \oplus Z_2, \phi_6)$, *using Equation (5) and* $\phi_6 = \{X \mapsto U \oplus Z_1, Y \mapsto U \oplus Z_2\}$.

Non-normalized narrowing steps such as

$$(X \oplus Y, id) \leadsto_{\phi_7} (Z, \phi_7), \text{ using Equation (5) and } \phi_7 = \{X \mapsto U \oplus U, Y \mapsto Z\}$$

are not in $VN^{\circlearrowleft}_{\mathscr{R}}$ because they are all subsumed by a variant narrowing step that computes the normalized version of the same substitution, e.g., $(Z, \phi_1) \leq_{\vec{E},B} (Z, \phi_7)$.

For a decomposition $(\Sigma, B, \vec{E})$, completeness of folding variant narrowing w.r.t. $\vec{E}$,B-normalized substitutions is proved in [Escobar et al., 2012, Theorem 4].

Chapter 3

Inspecting Maude Variants with GLINTS

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. 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. In this chapter we present GLINTS, a graphical tool for exploring variant narrowing computations in Maude that 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.

First, we present an overview of Maude's version 2.7.1, and how it implements the *folding variant narrowing strategy* in Section 3.1. In Section 3.2 an overall description of GLINTS is provided together with a leading example for describing GLINTS equational reasoning capabilities based on variant narrowing. We explain the core functionality of GLINTS and extra inspection features in Section 3.3. Finally, we provide a description of the tool implementation together with some experiments that assess its performance in Section 3.4.

3.1 Overview

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 (ACU). This novel equational unification relies on built-in generation of the set of variants of a term t [Durán et al. 2016]. 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* (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 omnipresent.

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 twenty 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 2018] 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.

The *GraphicaL Interactive Narrowing Tree Searcher*, GLINTS, is an inspecting tool for exploring variant computations in Maude and is publicly available from [GLINTS Website]. 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.

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.

3.2 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 3.1. *Assume that we test the FVP after replacing the variable declaration* `X:[NatSet]` *of the original specification with* `X:Nat`*:*

```
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 3.1 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 3.2.*

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 3.2); (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 [Alpuente et al. 2017a; Leuschel 2002]. Informally, a term t' embeds[1] 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 3.2 (by clicking on the symbol, its closest embedded ancestor gets also highlighted).

In Figure 3.2, note that we have interactively produced variants up to V_{10} and could continue generating variants indefinitely whereas the folding variant narrowing tree for the original `EXCLUSIVE-OR` theory stops at node V_6. Also note that some potential narrowing steps stemming from the nodes of Figure 3.2 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

[1] 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. 2017a].

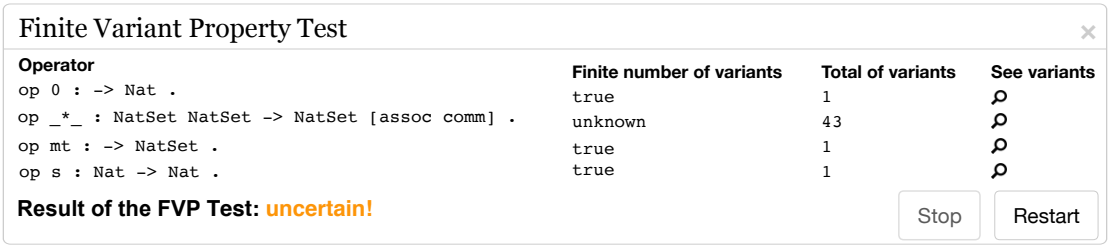


FIGURE 3.1: The FVP test for the modified non-FVP exclusive-or theory.

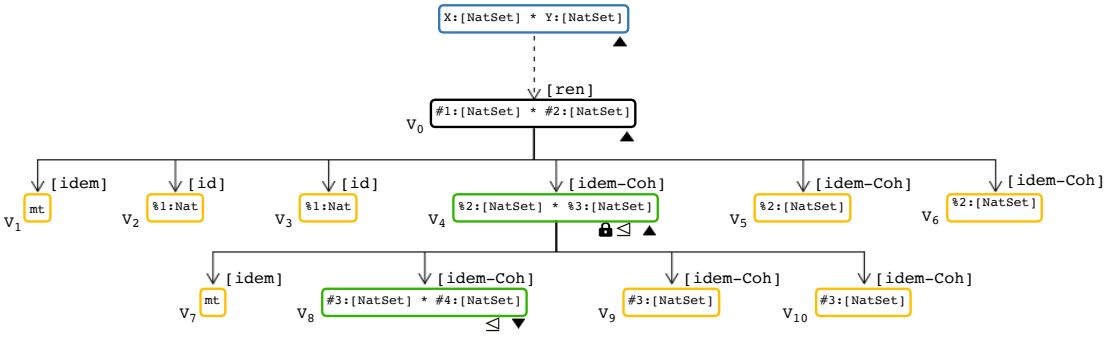


FIGURE 3.2: Inspecting variant computations of the modified non-FVP exclusive-or theory.

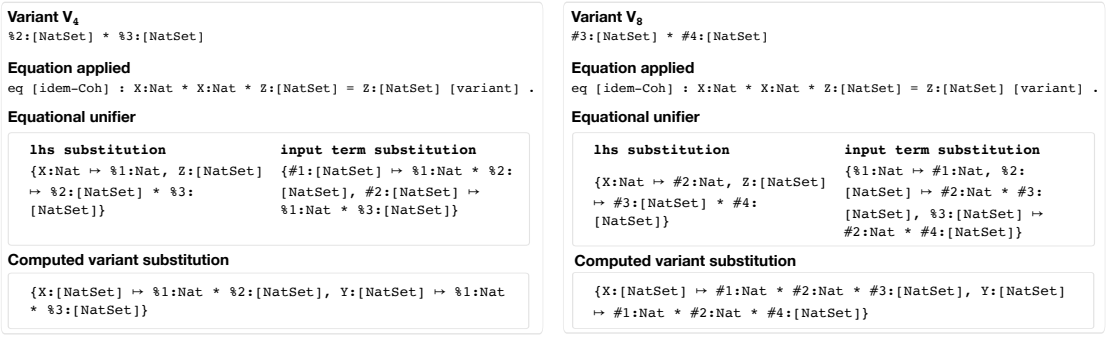


FIGURE 3.3: Comparison of nodes V_4 and V_8.

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 3.2 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 3.3, 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 finite variant property 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.

Example 3.2. *If we make the original specification and make the* `_*_` *symbol be associative, commutative, and with identity empty set element* `mt` *as shown below, then the theory does have*

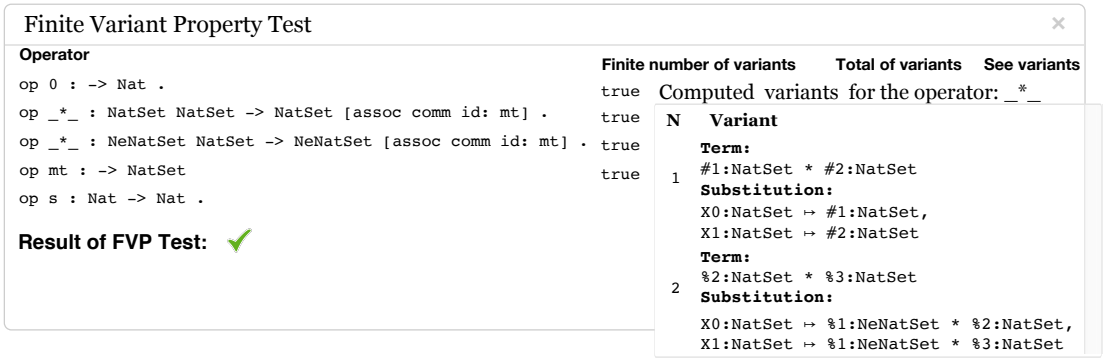


FIGURE 3.4: The FVP test for the newly modified exclusive-or theory with true verdict and variants for `_*_`.

FVP. This is shown in Figure 3.4; 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 .
  subsorts 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[2] of program calls that is used in the partial deduction (PD) 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. 1998b; Pettorossi and Proietti 1996]).

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

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. 1998a] and the partial evaluator Victoria for Maude equational theories of [Alpuente et al. 2017a], 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 PD, in the partial evaluator of [Alpuente et al. 2017a], *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 3.2 all the nodes are equationally closed; actually, they are either unnarrowable or a syntactic instance of the tree root.

3.3 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 finite variant property to be checked (as explained in Section 3.2).

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.

3.3.1 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.

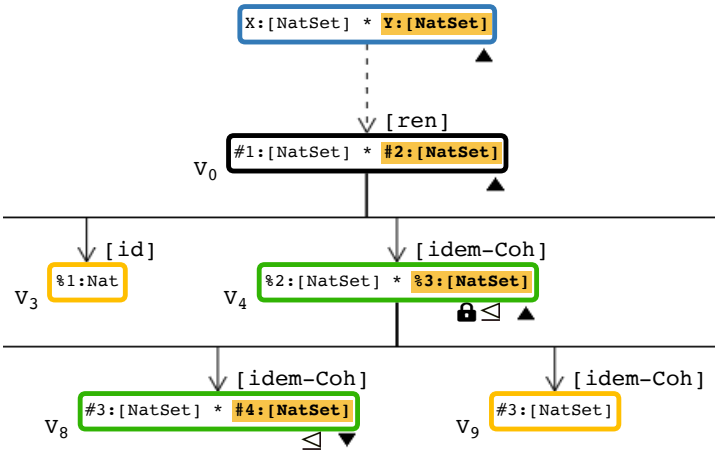


FIGURE 3.5: Result of the query "_ * ?" for the VNT of the non-FVP exclusive-or theory.

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 ACU, 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 3.2 highlights expressions `#2:[NatSet]`, `%3:[NatSet]`, and `#4:[NatSet]` in nodes V_0, V_4, and V_8, respectively, as shown in Figure 3.5.

3.3.2 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 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 $\equiv$ 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 n-th 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 variables of the term, as shown in Figure 3.6) 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

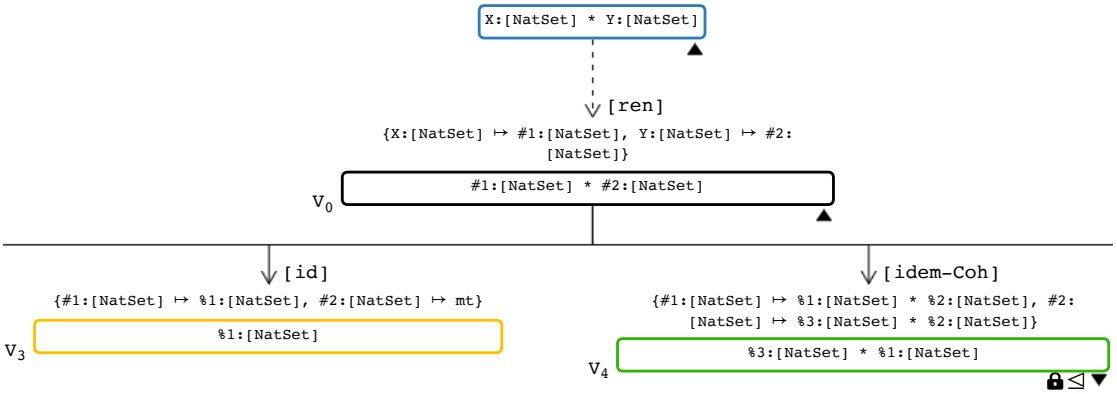


FIGURE 3.6: Enriched view showing equational unifiers for the original exclusive-or theory (fragment).

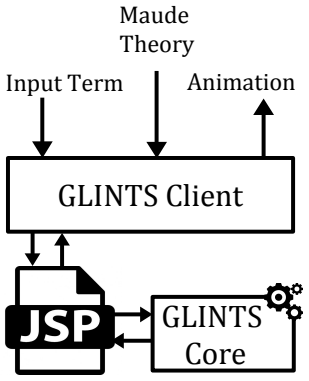


FIGURE 3.7: Architecture of GLINTS.

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 of the settings and detailed sessions can be found at [GLINTS Quick Start Guide].

3.4 Implementation

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

3.4.1 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 3.7. 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 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).

3.4.2 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[3] [Alpuente et al. 2016; Mau-Dev Website], 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 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 3.1 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 3.3GHz 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. 2017a]; and *Parser*, a generic parser for languages generated by simple, right regular grammars also from [Alpuente et al. 2017a]. 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.

[3]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.

	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

TABLE 3.1: Execution results of the `metaGetVariant` and `metaGetVariantExt` operations.

We have measured the `metaGetVariant` invocations on a statically compiled version of the 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 fiftieth 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 source code and experiments can be found at [GLINTS Experiments].

Chapter 4

Order-sorted Homeomorphic Embedding modulo Combinations of Associativity and/or Commutativity Axioms

The Homeomorphic Embedding relation has been amply used for defining termination criteria of symbolic methods for program analysis, transformation, and verification. However, homeomorphic embedding has never been investigated in the context of order-sorted rewrite theories that support symbolic execution methods *modulo* equational axioms. This chapter investigates the generalization of the symbolic homeomorphic embedding relation to order–sorted rewrite theories that may contain various combinations of associativity and/or commutativity axioms for different binary operators. We introduce five different formulations of order-sorted homeomorphic embedding modulo axioms and systematically measure their performance that we implement in Maude. We propose an order-sorted, equational homeomorphic embedding formulation $\check{\trianglelefteq}_B^{kosml}$ that runs up to 6 orders of magnitude faster than the original definition of the homeomorphic embedding modulo equational axioms $\trianglelefteq_B$ in [Alpuente et al. 2017a]. For this improvement in performance, we take advantage of Maude's powerful capabilities such as the efficiency of deterministic computations with equations versus non-deterministic computations with rewriting rules, or the use of non-strict definitions of the Boolean operators versus more speculative standard Boolean definitions [Clavel et al. 2007].

This chapter is organized as follows. Section 4.1 recalls the notion of pure homeomorphic embedding relation and summarizes our contributions. The pure (syntactic) homeomorphic embedding relation is formally described in Section 4.2. Section 4.3 recalls the (order-sorted) homeomorphic equational embedding relation of [Alpuente et al. 2017a] that extends the "syntactically simpler" homeomorphic embedding on nonground terms to the order-sorted case *modulo* equational axioms. Section 4.4 provides two *goal-driven* formulations for equational homeomorphic embedding: first, a calculus for embeddability goals that directly handles the algebraic axioms in the deduction system, and then a reachability-oriented characterization that cuts down the search space by taking advantage of pattern matching modulo associativity and commutativity axioms. Section 4.5 is concerned with an efficient meta-level formulation of equational homeomorphic embedding that relies on the classical flattening transformation that canonizes terms w.r.t. associativity and/or commutativity axioms (for instance, $1+(3+(2+0))$ gets flattened to $+(1,2,3)$). This formulation is optimized by replacing the classical Boolean

operators by short-circuit, strategic versions of these operators. In Section 4.6, two new optimizations of the algorithm are defined that can achieve significant speedup by anticipating failure, which is done by considering Maude's total order on terms and the compatibility of kinds for any possible instance of all subterms. In Section 4.7, we provide an experimental performance evaluation of the proposed formulations showing that we can efficiently deal with realistic embedding problems modulo axioms.

4.1 Overview

Homeomorphic Embedding is a control mechanism that is commonly used to ensure termination of symbolic methods and program optimization techniques. Homeomorphic embedding is a structural preorder relation under which a term t' is greater than (i.e., it embeds) another term t represented by $t \trianglelefteq t'$ if t can be obtained from t' by deleting some symbols of t'. For instance, $s(0+s(X))*s(X+Y)$ embeds $s(X)*s(Y)$. The usefulness of homeomorphic embedding for ensuring termination is given by the following well-known property of well-quasi-orderings: given a finite signature Σ, for every infinite sequence of terms $t_1, t_2, \ldots$, there exist $i < j$ such that $t_i \trianglelefteq t_j$. Therefore, if we iteratively compute a sequence $t_1, t_2, \ldots, t_n$, we can guarantee finiteness of the sequence by using the embedding as a whistle: whenever a new expression t_{n+1} is to be added to the sequence, we first check whether t_{n+1} embeds any of the expressions $t_i, i = 1, \ldots, n$ that are already in the sequence. If that is the case, the computation must be stopped because the whistle ($\trianglelefteq$) signals (potential) non-termination. Otherwise, t_{n+1} can be safely added to the sequence and the computation proceeds.

Recently, an experimental open platform has been developed in [Garavel et al. 2018] that allows the performance of functional and algebraic programming languages to be compared, including CafeOBJ, Clean, Haskell, LNT, LOTOS, Maude, mCRL2, OCaml, Opal, Rascal, Scala, SML, Stratego/XT, and Tom (see references in [Garavel et al. 2018]). In the top 5 of the more efficient tools, Maude ranks second after Haskell. This is remarkable for at least two reasons: (i) Maude is not a compiled language but runs under an interpreter; (ii) Maude has quite sophisticated features (subtype polymorphism, pattern matching modulo associativity, commutativity and identity, reflection, strategies, objects, etc.) that have no equivalent in Haskell or other functional languages.

In [Alpuente et al. 2017a], an extension of homeomorphic embedding modulo equational axioms was defined as a key component of Maude's symbolic partial evaluator, Victoria, which is based on (variant) narrowing [Escobar et al. 2012]. Unfortunately, the formulation in [Alpuente et al. 2017a] was done with a concern for simplicity in mind and degrades the tool performance because the proposed implementation of equational homeomorphic embedding did not scale well to realistic problems. This was not unexpected since other equational problems (such as equational matching, equational unification, or equational least general generalization) are typically much more costly than their corresponding "syntactic" counterparts and achieving efficient implementations has required years of significant investigation effort. Furthermore, the equational homeomorphic embedding relation $\trianglelefteq_B$ (modulo a set B of axioms) in [Alpuente et al. 2017a] did not consider types so that an embedding test such as `X:Bool` $\trianglelefteq_B$ `0 + suc(N:Nat)` succeeds.

The equational homeomorphic embedding relation $\trianglelefteq_B$ of [Alpuente et al. 2017a] was efficiently implemented in [Alpuente et al. 2018]. The novel contributions presented in this chapter are as follows:

1. A finer treatment of subtype hierarchies is provided that considers the *connected components* of sorts so that two variables whose respective types (sorts) are not related in the suptype hierarchy (e.g., `Bool` and `Nat`) are considered incomparable.

2. Extended definitions (and corresponding extended results) are formalized dealing with sorts and subsorts in such a finer way.

3. Two novel optimizations are proposed that can achieve significant improvements in performance. Each optimization is formulated as a pruning rule that anticipates failure by considering Maude's total order on terms and the compatibility of kinds for any possible instance of all subterms.

4. A full treatment of all aspects is given, including more examples and detailed explanations of the key notions, together with full formal proofs of all technical results.

5. The implementation and experimental section have been improved in two ways: (1) the prototype implementation itself has been advanced with a new embedding test $\check{\trianglelefteq}_B^{kosml}$ that implements the novel optimizations; and (2) new examples that benchmark these optimizations with increasingly larger terms are provided. One important difference with the smaller set of benchmarks in [Alpuente et al. 2018] is that much bigger terms can be now handled by the tool.

4.2 Pure homeomorphic embedding

The pure (syntactic) homeomorphic embedding relation known from term algebra [Kruskal 1960] was introduced by Dershowitz for fixed-arity symbols in [Dershowitz 1979] and for variable-arity symbols in [Dershowitz and Jouannaud 1990]. In the following, we consider only fixed-arity symbols.

Definition 4.1 (Homeomorphic embedding, [Dershowitz 1979])**.** The homeomorphic embedding relation $\trianglelefteq$ over $\mathcal{T}_\Sigma$ is defined as follows:

$$\frac{\exists i\in\{1,\ldots,n\} : s \trianglelefteq t_i}{s \trianglelefteq f(t_1,\ldots,t_n)} \qquad\qquad \frac{\forall i\in\{1,\ldots,n\} : s_i \trianglelefteq t_i}{f(s_1,\ldots,s_n) \trianglelefteq f(t_1,\ldots,t_n)}$$

with $n \geq 0$.

Roughly speaking, the left inference rule deletes subterms, while the right inference rule deletes context. We write $s \trianglelefteq t$ if s is derivable from t using the above rules. When $s \trianglelefteq t$, we say that s is (syntactically) *embedded* in t (or t syntactically *embeds* s). Note that $\equiv \subseteq \trianglelefteq$, where $\equiv$ denotes syntactic identity.

A well-quasi-ordering (wqo) $\preceq$ is a transitive and reflexive binary relation such that, for any infinite sequence of terms $t_1, t_2, \ldots$ with a finite number of operators, there exist i, j with $i < j$ and $t_i \preceq t_j$.

Theorem 4.2 (Tree Theorem, [Kruskal 1960])**.** *The embedding relation $\trianglelefteq$ is a well-quasi-ordering on $\mathcal{T}_\Sigma$.*

4.2.1 Mechanizing the Homeomorphic Embedding

The derivability relation given by $t \trianglelefteq t'$ is mechanized in [Middeldorp and Gramlich 1995] by introducing a term rewriting system $Emb(\Sigma)$ that is used to rewrite t' to t, i.e., $t' \to^*_{Emb(\Sigma)} t$. Similarly to Definition 4.1, order-sorted signatures are not considered.

Definition 4.3 (Rewrite theory for $\trianglelefteq$, [Middeldorp and Gramlich 1995])**.** Let Σ be an unsorted signature. Homeomorphic embedding can be decided by a rewrite theory $Emb(\Sigma) = (\Sigma, \emptyset, R)$ such that R consists of rewrite rules of the form

$$f(x_1, \cdots, x_n) \to x_i$$

where $f \in \Sigma$ is a function symbol of arity $n \geq 1$ and $i \in \{1, \cdots, n\}$.

Lemma 4.4 ([Middeldorp and Gramlich 1995])**.** *Given an unsorted signature Σ and two terms $t_1, t_2 \in \mathcal{T}_\Sigma$, we have $t \trianglelefteq t'$ iff $t' \to^*_{Emb(\Sigma)} t$.*

Definition 4.1 can be applied to terms of $\mathcal{T}_\Sigma(\mathcal{X})$ by simply regarding the variables in terms as constants. However, this definition cannot be used when existentially quantified variables are considered (as in logic programming or symbolic execution). The following definition from [Leuschel 1998b; Sørensen and Glück 1995] adapts the pure (syntactic) homeomorphic embedding from [Dershowitz and Jouannaud 1990] by adding a simple treatment of logical variables where all variables are treated as if they were identical, which is enough for many symbolic methods such as the partial evaluation of [Alpuente et al. 2017a]. Some extensions of $\trianglelefteq$ dealing with varyadic symbols and infinite signatures are investigated in [Leuschel 2002].

4.2.2 Symbolic Homeomorphic Embedding

The homeomorphic embedding is commonly applied not only to ground terms but also to terms with variables. The extension to variables is given by [Leuschel 1998b].

Definition 4.5 (Symbolic homeomorphic embedding, [Leuschel 1998b])**.** The extended homeomorphic embedding relation $\trianglelefteq$ over $\mathcal{T}_\Sigma(\mathcal{X})$ is defined in Figure 4.1, where the Variable inference rule allows dealing with free (unsorted) variables in terms, while the Diving and Coupling inference rules are similar to the pure (syntactic) homeomorphic embedding definition.

Variable	Diving	Coupling
$\dfrac{}{x \trianglelefteq y}$	$\dfrac{\exists i \in \{1,\ldots,n\} : s \trianglelefteq t_i}{s \trianglelefteq f(t_1,\ldots,t_n)}$	$\dfrac{\forall i \in \{1,\ldots,n\} : s_i \trianglelefteq t_i}{f(s_1,\ldots,s_n) \trianglelefteq f(t_1,\ldots,t_n)}$

FIGURE 4.1: Symbolic homeomorphic embedding

For instance, given variables X and Y, $X \trianglelefteq g(Y)$, $g(X) \trianglelefteq f(g(Y))$, and $f(X) \trianglelefteq f(g(X))$ but $g(0) \not\trianglelefteq g(X)$. Note that the embedding relation $\trianglelefteq$ does not subsume the relative generality ordering $\leq$, e.g., $f(X) \not\trianglelefteq f(g(0))$ even if $f(g(0))$ is an instance of $f(X)$. Also note that the embedding relation $\trianglelefteq$ is not closed under instantiation, e.g., the instance of $g(X) \trianglelefteq f(g(Y))$ with substitution $\{X \mapsto 0, Y \mapsto 0\}$ holds, in symbols $g(0) \trianglelefteq f(g(0))$, whereas the instance of $g(X) \trianglelefteq f(g(Y))$ with substitution $\{X \mapsto g(Z)\}$ does not hold, in symbols $g(g(Z)) \not\trianglelefteq f(g(Y))$.

Theorem 4.6 ([Leuschel 1998b; Sørensen and Glück 1995])**.** *The embedding relation $\trianglelefteq$ is a well-quasi-ordering on $\mathcal{T}_\Sigma(\mathcal{X})$.*

4.2.3 Adding sorts and subsorts

When we are interested in adding sorts and subsorts, the extension of the homeomorphic embedding to the order-sorted setting is not completely trivial. Let us provide a motivating example.

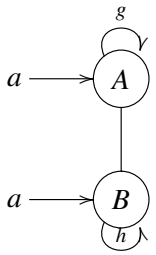


FIGURE 4.2: Signature graph of Example 4.1

Example 4.1. *Consider the order-sorted signature depicted in Figure 4.2 that defines two sorts A and B, with $B < A$. Also, consider an overloaded constant a for sorts A and B, and two operators $g : A \to A$ and $h : B \to B$. Strictly speaking, the no ad-hoc overloading requirement would rule out ad-hoc overloaded constants such as a. However, this requirement can be relaxed in a natural and easy way by allowing constants to be qualified by their sort. For instance, in Maude this is done by enclosing them in parentheses followed by a dot and the sort name. For example, we can disambiguate the term a by writing either $(a).A$ or $(a).B$.*

In an unsorted setting, where sorts are simply disregarded, both the embedding goals $a \trianglelefteq g(a)$ and $a \trianglelefteq h(a)$ hold.

In a many-sorted setting, where sorts are considered but subsort hierarchies are disregarded, it is immediate to extend the unsorted homeomorphic embedding relation $\trianglelefteq$ of Definition 4.1 to the many-sorted case. On the one hand, two constants $(a).A$ and $(a).B$ that belong to different sorts are distinct and hence incomparable by embedding. On the other hand, since we assume there is no ad-hoc overloading (except for constants), no extra check must be added to the Coupling rule because no two symbols f (of different sorts) exist, except for constants, while the Diving rule should be applied regardless of sorts. Thus, by abusing notation, $(a).A \not\trianglelefteq (a).B$ and $(a).B \not\trianglelefteq (a).A$, and hence $(a).A \trianglelefteq g((a).A)$ and $(a).B \trianglelefteq h((a).B)$, whereas $(a).B \not\trianglelefteq g((a).A)$ and $(a).A \not\trianglelefteq h((a).B)$.

Extending $\trianglelefteq$ to the order-sorted case is also easy in kind-complete signatures, where each constant has a kind (and so does any ground term). Therefore, by lifting all symbol declaration to the kind level, two constants $(c).s_1$ and $(c).s_2$ are only incomparable if the kinds $[s_1]$ and $[s_2]$ are different. By abusing notation, this means that all of the four embedding goals $(a).A \trianglelefteq g((a).A)$, $(a).B \trianglelefteq h((a).B)$, $(a).B \trianglelefteq g((a).A)$, and $(a).A \trianglelefteq h((a).B)$ hold (because $[A] = [B]$), whereas none of them would hold if we substitute A (resp. B) by a new sort C such that $[A] \neq [C]$ (resp. $[B] \neq [C]$).

Let us now consider the problem of extending Definition 4.5 to the order-sorted case, which requires to reason about variables. Fortunately, in kind-complete signatures every variable has a kind (and so does any term). Following Maude syntax, let us qualify any variable X by appending its name with a semicolon followed by the sort name, e.g. $X{:}A$. Then, given sorts A, B, and C, with $B < A$, variables $X{:}A$ and $X{:}B$ are in the same kind, $[A]$, whereas $X{:}A$ and $X{:}C$ are in different kinds, $[A]$ and $[C]$, respectively.

Let us define the order-sorted symbolic homeomorphic embedding relation as follows.

Definition 4.7 (Order-sorted symbolic homeomorphic embedding)**.** The order-sorted symbolic homeomorphic embedding relation $\overset{\vee}{\trianglelefteq}$ over $\mathcal{T}_\Sigma(\mathcal{X})_s$ is defined in Figure 4.3, where the Variable inference rule allows dealing with free order-sorted variables in terms while the Diving and Coupling inference rules are similar to Definition 4.5.

Variable	Diving	Coupling
$\frac{\lceil x\rceil=\lceil y\rceil}{x \overset{\vee}{\trianglelefteq} y}$	$\frac{\exists i\in\{1,\ldots,n\} : s \overset{\vee}{\trianglelefteq} t_i}{s \overset{\vee}{\trianglelefteq} f(t_1,\ldots,t_n)}$	$\frac{\forall i\in\{1,\ldots,n\} : s_i \overset{\vee}{\trianglelefteq} t_i}{f(s_1,\ldots,s_n) \overset{\vee}{\trianglelefteq} f(t_1,\ldots,t_n)}$

FIGURE 4.3: Order-sorted extended homeomorphic embedding

It is worth noting that, while it seems natural to consider that $X{:}A \overset{\vee}{\trianglelefteq} Y{:}B$ for the case when A is a subsort of B (which holds because $[A] = [B]$, hence $\lceil x\rceil = \lceil y\rceil$), the practical usefulness of having $X{:}A \overset{\vee}{\trianglelefteq} Y{:}B$ is less evident for the case when B is a subsort of A (which holds for the very same reason). However, consider an overloaded operator g, with $g : A \to A$ and $g : B \to B$, with $A > B$. In a context of symbolic execution where logical variables are considered, it could be the case of having a computation sequence $(t_1,\ldots t_i,\ldots,t_j,\ t_{j+1}\ldots)$, with $t_i = g(X{:}A)$, $t_j = g(Y{:}B)$, and $t_{j+1} = g(X{:}A)$, where t_{j+1} derives from t_j by a symbolic execution step (e.g., think of a narrowing step from $g(Y{:}B)$ by using a program rule $g(g(x{:}A)) \to g(x{:}A)$). Hence, the fact that $g(X{:}A) \overset{\vee}{\trianglelefteq} g(X{:}B)$ allows the risk of non-termination to be detected at t_j, i.e., before generating t_{j+1}.

Example 4.2. *Consider the order-sorted signature depicted in Figure 4.4 which defines three sorts A, B, and C, with $B < A$. Also, consider a constant a of sort A, an operator $f : A \to B$, an operator $d : B \to A$, the (subsort) overloaded operator g for sorts A and B, with $g : A \to A$ and $g : B \to B$, a constant c of sort C, and an operator $h : C \to A$. Note that there are two different kinds $[A]$ and $[C]$, where $[A]$ contains sorts A and B,*

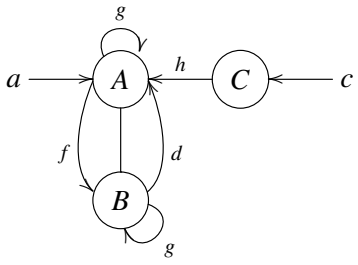


FIGURE 4.4: Signature graph of Example 4.2

Now consider three variables X:A, Y:B, and Z:C. By applying first the coupling rule and then the diving rule, the embedding goal $g(Y) \overset{\vee}{\trianglelefteq} g(f(X))$ holds because variables X and Y are in the same kind. However, $g(Y) \overset{\vee}{\trianglelefteq} g(h(Z))$ (resp. $g(X) \overset{\vee}{\trianglelefteq} g(h(Z))$) does not hold because Y (resp. X) and Z are in different kinds.

Note that the embedding goal $g(X) \overset{\vee}{\trianglelefteq} g(Y)$ does also hold even though $g(X)$ is of sort A, $g(Y)$ is of sort B, and $B < A$. Similarly, the embedding goal $Y \overset{\vee}{\trianglelefteq} d(X)$ does also hold even if the term $d(X)$ is not well-typed but admissible in Maude[1]; it belongs to the kind $[A]$ but not to sorts A and

[1]This is because we are assuming that, as in Maude, each signature is extended to a kind-complete one (recall Chapter 2).

B. Actually, $d(X)$ has well-typed instances such as $d(Y)$, with the sort-decreasing substitution $\{X \mapsto Y\}$. Furthermore, the embedding goal $Z \overset{\vee}{\trianglelefteq} d(X)$ does not hold, which is consistent with the fact that X cannot be instantiated with a term of sort C.

Similarly to $\trianglelefteq$, it is immediate to see that the order-sorted symbolic embedding relation $\overset{\vee}{\trianglelefteq}$ is a well-quasi-ordering on the set of terms $\mathscr{T}_\Sigma(\mathscr{X})$.

Theorem 4.8. *The embedding relation $\overset{\vee}{\trianglelefteq}$ is a well-quasi-ordering on $\mathscr{T}_\Sigma(\mathscr{X})$.*

Proof. The proof is similar to [Leuschel 1997]. We need the following concept from [Dershowitz and Jouannaud 1990] that we adapt to fixed-arity symbols. Let $\overset{<}{\sim}$ be a relation on a set $\mathscr{S}$ of symbols. Then the embedding extension of $\overset{<}{\sim}$ is a relation $\overset{<}{\sim}_{emb}$ on terms, constructed (only) from the symbols in $\mathscr{S}$, which is inductively defined as follows:

1. $t \overset{<}{\sim}_{emb} f(t_1,...,t_n)$ if $t \overset{<}{\sim}_{emb} t_i$ for some i;
2. $f(s_1,\ldots,s_n) \overset{<}{\sim}_{emb} g(t_1,\ldots,t_n)$ if $f \overset{<}{\sim} g$ and $\forall i \in \{1,\ldots,n\} : s_i \overset{<}{\sim}_{emb} t_i$.

We define the relation $\overset{<}{\sim}$ on the set $\mathscr{S} = (\Sigma \cup \mathscr{X})$ of symbols as the least relation satisfying:

1. $x \overset{<}{\sim} y$ if $x \in \mathscr{X}$, $y \in \mathscr{X}$, and $\lceil x \rceil = \lceil y \rceil$;
2. $f \overset{<}{\sim} f$ if $f \in \Sigma$.

This relation is a wqo on $\mathscr{S}$ because Σ is finite. Therefore, by the Higman-Kruskal's theorem (see e.g., [Dershowitz and Jouannaud 1990]), its embedding extension to terms, $\overset{<}{\sim}_{emb}$, (which is by definition identical to $\overset{\vee}{\trianglelefteq}$) is a wqo on $\mathscr{T}_\Sigma(\mathscr{X})$. □

4.2.4 Getting rid of variables

In the following, we show how it is possible to reformulate the homeomorphic embedding calculus for order-sorted terms with variables without the hassle of dealing with variables explicitly.

We need the following notation. Given the set $\mathscr{K} = \{\lceil \mathsf{s} \rceil \mid \mathsf{s} \in \mathsf{S}\}$ of all kinds in the equational theory (Σ, E), for each kind $k \in \mathscr{K}$ we consider a fresh constant symbol $\sharp_k$. Let $\Pi_\mathscr{K}$ be the set of all such symbols, $\Pi_\mathscr{K} = \{\sharp_k \mid k \in \mathscr{K}\}$. Given an order-sorted signature Σ, we define $\Sigma^\sharp$ as the extension of Σ with the constants in $\Pi_\mathscr{K}$. For a term $t \in \mathscr{T}_\Sigma(\mathscr{X})$, let $t^\sharp \in \mathscr{T}_{\Sigma^\sharp}$ denote the (ground) instance of t where every variable x of sort s is replaced by the corresponding $\sharp_k$, with $k = \lceil \mathsf{s} \rceil$ being the kind of sort s.

The following result extends [Alpuente et al., 2018, Lemma 1] to the order-sorted case.

Lemma 4.9 (Variable-less characterization of $\overset{\vee}{\trianglelefteq}$)**.** *Given two terms $t_1, t_2 \in \mathscr{T}_\Sigma(\mathscr{X})$, we have $t_1 \overset{\vee}{\trianglelefteq} t_2$ iff $t_1^\sharp \blacktriangleleft t_2^\sharp$ where $t_1^\sharp, t_2^\sharp \in \mathscr{T}_{\Sigma^\sharp}$.*

Proof. Immediate by structural induction. The only relevant case is the base case, i.e., when the Variable inference rule $\frac{\lceil x \rceil = \lceil y \rceil}{x \overset{\vee}{\trianglelefteq} y}$ is applied. In the grounded homeomorphic embedding problems $t_1^\sharp \blacktriangleleft t_2^\sharp$ and $t_1^\sharp \trianglelefteq t_2^\sharp$, the corresponding case is $\sharp_k \trianglelefteq \sharp_k$ for the same kind k, and the result follows directly. □

By abuse of notation, from now on, we will indistinctly consider either terms with variables or ground terms with corresponding $\sharp_k$ symbols, whenever one formulation is simpler than the other.

Example 4.3. *Consider again the order-sorted signature of Example 4.2. Also consider the embedding goals* $g(Y) \check{\trianglelefteq} g(f(X))$ *and* $g(Y) \check{\trianglelefteq} g(h(Z))$*, which were respectively proved to hold (because variables X and Y are in the same kind, i.e.,* $\lceil X \rceil = \lceil Y \rceil = [A]$*) and not to hold (because Y and Z are in different kinds). For the corresponding grounded goals, it is immediate to see that* $g(\sharp_{[A]}) \trianglelefteq g(f(\sharp_{[A]}))$ *holds whereas* $g(\sharp_{[A]}) \trianglelefteq g(h(\sharp_{[C]}))$ *does not hold.*

4.3 Homeomorphic embedding modulo equational axioms

Let us extend the order-sorted homeomorphic embedding relation on nonground terms $\check{\trianglelefteq}$ to the case when we compare terms *modulo* a set of axioms B.

In [Alpuente et al. 2017a], an equational extension of the "syntactically simpler" homeomorphic embedding relation $\trianglelefteq$ on nonground terms, called the B–embedding relation $\trianglelefteq_B$ (or embedding modulo B), was defined as follows: $\trianglelefteq_B \equiv (\overset{ren}{=}_B).(\trianglelefteq).(\overset{ren}{=}_B)$, where $v \overset{ren}{=}_B v'$ iff there is a renaming substitution σ for v' such that $v =_B v'\sigma$. We define the corresponding order-sorted extension $\check{\trianglelefteq}_B$ of $\trianglelefteq_B$ in the natural way.

Definition 4.10 ((Order-sorted) homeomorphic embedding modulo B)**.** The order-sorted B–embedding relation $\check{\trianglelefteq}_B$ is $(\overset{ren}{=}_B).(\check{\trianglelefteq}).(\overset{ren}{=}_B)$.

Example 4.4. *Consider the following rewrite theory (written in Maude syntax) that defines the signature of natural numbers. The defined sort hierarchy has top sort* `Nat` *and (disjoint) subsorts* `Zero` *and* `NzNat` *(for non-zero natural numbers). The sort* `Nat` *is generated from the constant* `0` *(of sort* `Zero`*) and the successor operator* `suc`[2] *(of sort* `NzNat`*). We also define the associative and commutative natural addition operator symbol* `_+_` *for sort* `Nat` *but add two extra subsort-overloaded definitions.*

```
fmod NAT is
  sorts Zero NzNat Nat .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero .
  op suc : Nat -> NzNat .
  op _+_ : Zero Zero -> Zero   [assoc comm] .
  op _+_ : NzNat Zero -> NzNat [assoc comm] .
  op _+_ : Nat Nat -> Nat      [assoc comm] .
endfm
```

Then, we have `1 + X:Nat` $\check{\trianglelefteq}_B$ `Y:Nat + (1 + 2)` *because* `Y:Nat + (1 + 2)` *is equal to* `1 + (2 + Y:Nat)` *modulo the associativity and commutativity of* `_+_`*, and* `1 + X:Nat` *is homeomorphically embedded into* `1 + (2 + Y:Nat)`*. Similarly,* `1 + X:Zero` $\check{\trianglelefteq}_B$ `Y:Nat + (1 + 2)` *and* `1 + X:NzNat` $\check{\trianglelefteq}_B$ `Y:Nat + (1 + 2)` *hold. Note that* `suc(X:Nat)` $\check{\trianglelefteq}_B$ `suc(Y:Zero)` *holds despite the fact that* `Zero < Nat`*; this is a deliberate decision as explained in Example 4.2.*

[2]For simplicity, we represent natural numbers in normal decimal notation; e.g., `2` for `suc(suc(0))`.

The following result extends Kruskal's Tree Theorem for the equational theories B considered in this thesis, which we restrict to *class–finite* equational theories. B is called *class-finite* if all of the B-equivalence classes of terms in the quotient term algebra $\mathcal{T}_\Sigma(\mathcal{X})/=_B$ are finite. This includes the class of permutative equational theories: an equational theory $\mathscr{E}$ is permutative if for all terms t, t', the fact that $t =_\mathscr{E} t'$ implies that the terms t and t' contain the same symbols with the same number of occurrences [Bürckert et al. 1989]. Permutative theories include any theory with any combination of symbols obeying any combination of associativity and commutativity axioms.

Theorem 4.11. *For class-finite theories, the embedding relation $\check{\trianglelefteq}_B$ is a well-quasi-ordering on the set $\mathcal{T}_\Sigma(\mathcal{X})$ for finite Σ.*

Proof. A binary relation $\succ$ is Noetherian (i.e., well-founded) on a set X if and only if its dual relation $\preceq$ (defined as $u \preceq v$ iff $u \not\succ v$) is well-founded on X, i.e., in every sequence $(x_i)_{i\in\mathbb{N}}$ of elements of X, there exist $i < j$ such that $x_i \preceq x_j$ [Melliès 1998]. Since $\check{\trianglelefteq}$ is a wqo (by Theorem 4.8), the result follows for class-finite theories from the fact that $>_B$ is well-founded [Bürckert et al. 1989], and $\check{\trianglelefteq}$ is compatible with $(\overset{ren}{=_B})$. □

Similarly to Lemma 4.9, in the equational case, we can also get rid of variables and consider only ground terms.

Lemma 4.12. *Given two terms $t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$, we have $t_1 \check{\trianglelefteq}_B t_2$ iff $t_1^\sharp \check{\trianglelefteq}_B t_2^\sharp$ where $t_1^\sharp, t_2^\sharp \in \mathcal{T}_{\Sigma^\sharp}$.*

Function symbols with variable arity are sometimes seen as associative operators. Let us briefly discuss by means of an example the homeomorphic embedding modulo axioms $\check{\trianglelefteq}_B$ of Definition 4.10 when compared with the variadic extension $\trianglelefteq^v$ of Definition 4.1 as given in [Dershowitz and Jouannaud 1990]:

$$\text{Diving}\quad \frac{\exists i\in\{1,\ldots,n\} : s \trianglelefteq^v t_i}{s \trianglelefteq^v f(t_1,\ldots,t_n)} \qquad \text{Coupling}\quad \frac{\forall i\in\{1,\ldots,m\} : s_i \trianglelefteq^v t_{j_i}, \text{with } 1\le j_1<j_2<\cdots<j_m\le n}{f(s_1,\ldots,s_m) \trianglelefteq^v f(t_1,\ldots,t_n)}$$

Example 4.5. *Consider a variadic version of the addition symbol $+$ of Example 4.4 that allows any number of natural numbers to be used as arguments; for instance, $+(1,2,3)$. On the one hand, $+(1) \trianglelefteq^v +(1,2,3)$ whereas $+(1) \not\check{\trianglelefteq}_B +(1,2,3)$, with B consisting of the associativity and commutativity axioms for the operator $+$ (actually, $+(1)$ is ill-formed). On the other hand, we have both $+(1,2) \trianglelefteq^v +(1,0,3,2)$ and $+(1,2)\check{\trianglelefteq}_B +(1,0,3,2)$. This is because any well-formed term that consists of the addition (in any order) of the constants 0, 1, 2, and 3 (for instance, $+(+(1,0),+(3,2))$ can be given a flat representation $+(1,0,2,3)$. Note that there are many other equivalent terms, e.g., $+(+(1,2),+(3,0))$ or $+(+(1,+(3,2)),0)$, all of which are represented by the flattened term $+(0,1,2,3)$. Actually, because of the associativity and commutativity of symbol $+$, flattened terms like $+(1,0,2,3)$ can be further simplified into a single*[3] canonical representative $+(0,1,2,3)$, *hence also $+(1,2)\check{\trianglelefteq}_B +(0,1,2,3)$. A more detailed explanation of flat terms can be found in Section 4.5. However, note that $+(2,1)\check{\trianglelefteq}_B +(1,0,3,2)$ but $+(2,1) \not\trianglelefteq^v +(1,0,3,2)$ because the relation $\trianglelefteq^v$ does not consider the commutativity of symbol $+$.*

Roughly speaking, in the worst case, the homeomorphic embedding modulo axioms B of Definition 4.10, $t \check{\trianglelefteq}_B t'$, amounts to considering all of the elements in the B-equivalence classes

[3] Maude uses a term lexicographic order for the arguments of flattened terms [Eker 2002].

of t and t' and then checking for standard homeomorphic embedding, $u \mathrel{\breve{\trianglelefteq}} u'$, every pair u and u' of such terms, one term from each class. Unfortunately, the enumeration of all terms in a B-equivalence class is impractical, as shown in the following example.

Example 4.6. *Consider the AC binary symbol $+$ of Example 4.4 and the terms $t = +(1,2)$ and $t' = +(2,+(3,1))$. The AC-equivalence class of t contains two terms whereas the AC-equivalence class of t' contains 12 terms. This implies checking 24 embedding problems $u \mathrel{\breve{\trianglelefteq}} u'$ in order to decide $t \mathrel{\breve{\trianglelefteq}}_{AC} t'$, in the worst case. Moreover, we know a priori that half of these embedding tests will fail (those in which 1 and 2 occur in different order in u' and u; for instance $u' = +(1,+(2,3))$ and $u = +(2,1)$.*

A more effective rewriting characterization of $\mathrel{\breve{\trianglelefteq}}_B$ can be achieved by lifting Definition 4.3 to both the order-sorted and the *modulo* case in a natural way. However, ill-formed terms can be produced by naïvely applying the rules $f(x_1,\ldots,x_n) \to x_i$ of Definition 4.3 to typed (i.e., order-sorted) terms. For example, "$(0 \leq 1)$ `or true`" $\to$ "`0 or true`".

In the order-sorted context, we can overcome this drawback as follows. We extend Σ to a new signature $\Sigma^{\mathscr{U}}$ by adding a new top sort $\mathscr{U}$ that is bigger than all other sorts. Now, for each $f : A_1,\ldots,A_n \to A$ in Σ, we add the rules $f(x_1{:}\mathscr{U},\ldots,x_n{:}\mathscr{U}) \to x_i{:}\mathscr{U}$, $1 \leq i \leq n$. In this way, rewriting with $\to^*_{Emb(\Sigma^{\mathscr{U}})/B}$ becomes a relation between well-formed $\Sigma^{\mathscr{U}}$-terms, as first proposed in [Alpuente et al. 2017a].

The order-sorted homeomorphic embedding modulo B can be decided by the following rewrite theory that extends the definition in [Alpuente et al. 2017a] with sorts.

Definition 4.13 (Rewrite theory for $\mathrel{\breve{\trianglelefteq}}_B$)**.** Let Σ be an order-sorted signature and B be a set of axioms. Let us introduce the following signature transformation $\Sigma \ni (f : \mathsf{s}_1 \ldots \mathsf{s}_n \to \mathsf{s}) \mapsto (f : \mathscr{U} \overset{n}{\ldots} \mathscr{U} \to \mathscr{U}) \in \Sigma^{\mathscr{U}}$, where $\mathscr{U}$ conceptually represents a universal supersort of all sorts in Σ. Since there is no ad-hoc overloading, any term $t \in \mathscr{T}_\Sigma$ is well-typed for $\Sigma^{\mathscr{U}}$.

We define the rewrite theory $Emb(\Sigma) = (\Sigma, B, R)$ such that R consists of all rewrite rules

$$f(x_1{:}\mathscr{U},\ldots,x_n{:}\mathscr{U}) \to x_i{:}\mathscr{U}$$

for each $f : A_1,\ldots,A_n \to A$ in Σ and $i \in \{1,\ldots,n\}$.

Proposition 4.14. *Given Σ, B, and terms $t, t' \in \mathscr{T}_\Sigma(\mathscr{X})$, $t \mathrel{\breve{\trianglelefteq}}_B t'$ iff $t'^\sharp \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$.*

Proof. By Lemma 4.12, $t \mathrel{\breve{\trianglelefteq}}_B t'$ iff $t^\sharp \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$. We prove $t^\sharp \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$ iff $t'^\sharp \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$.

($\Rightarrow$) We prove that $t^\sharp \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$ implies $t'^\sharp \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$. By Definition 4.10, there are two terms w, w' such that $t^\sharp =_B w$, $t'^\sharp =_B w'$, and $w \mathrel{\breve{\trianglelefteq}} w'$. By Lemma 4.9 $w \mathrel{\breve{\trianglelefteq}} w'$ iff $w \mathrel{\blacktriangleleft} w'$. By Lemma 4.4, $w \mathrel{\blacktriangleleft} w'$ implies $w' \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})} w$. And this implies $t'^\sharp \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$.

($\Leftarrow$) We prove that $t'^\sharp \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$ implies $t^\sharp \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$ by induction on the length k of the rewriting sequence. If $k = 0$, then there is a constant a such that $t'^\sharp = a = t^\sharp$ and $a \to^*_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} a$. Thus, $a \mathrel{\breve{\trianglelefteq}}_B a$ by using the Coupling inference rule of Figure 4.3 with $n = 0$. If $k > 0$, then there exists w such that $t'^\sharp \to^{k-1}_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} w \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} t^\sharp$. Given the form $f(x_1{:}\mathscr{U},\ldots,x_n{:}\mathscr{U}) \to x_i{:}\mathscr{U}$ of the rewrite rules in $Emb((\Sigma^\sharp)^{\mathscr{U}})$, we have that $w =_B C[f(t_1,\ldots,t_n)]$ and there exists

$i \in \{1,\ldots,n\}$ s.t. $C[t_i] =_B t^\sharp$. Hence, it is immediate to prove $t^\sharp \mathrel{\breve{\trianglelefteq}}_B w$ by a straightforward combination of several applications of the Coupling inference rule to remove the context $C[\,]$ from both terms, $t^\sharp$ and w, one application of the Diving inference rule to extract t_i from $f(t_1,\ldots,t_n)$ in w, and the remaining applications of the Coupling inference rule that are needed to recognize the embedding of t_i inside $t^\sharp$. Now, by induction hypothesis we also have that $w \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$, which yields $t^\sharp \mathrel{\breve{\trianglelefteq}}_B w \mathrel{\breve{\trianglelefteq}}_B t'^\sharp$. □

Example 4.7. *Consider the order-sorted signature for natural numbers of Example 4.4. Let us represent by sort* `U` *in Maude the unique (top) sort of the transformed signature:*

```
fmod NAT-U is
  sort U .
  op 0 : -> U .
  op suc : U -> U .
  op _+_ : U U -> U [assoc comm] .
endfm
```

Since we have no ad-hoc overloading, there is no need to transform a ground term of module `NAT` *into a ground term of module* `NAT-U` *since any ground term of sorts* `Zero`, `NzNat`, *and* `Nat` *is well typed in the transformed signature. The associated rewrite theory* $Emb((\Sigma^\sharp)^{\mathscr{U}})$ *contains the following two rules for the operator* $+$*:*

$$+(x_1{:}U, x_2{:}U) \to x_1{:}U$$
$$+(x_1{:}U, x_2{:}U) \to x_2{:}U$$

However, when the rules of $Emb((\Sigma^\sharp)^{\mathscr{U}})$ *are used for rewriting modulo the commutativity of symbol* $+$ *(as in Maude), in practice, we can get rid of one of the two rules above since only one of them is required.*

Example 4.8. *Following Example 4.6, instead of comparing pairwisely all terms in the equivalence classes of* t *and* t'*, we use the rewrite rule* $+(x_1{:}U, x_2{:}U) \to x_2{:}U$ *to prove the rewrite step* $+(2,+(3,1)) \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} +(2,1)$*, and finally, we check that* $+(2,1) =_B +(1,2)$*. Recall that* B *contains associativity and commutativity of* $+$*. However, there are six alternative rewriting steps stemming from the initial term* $+(2,+(3,1))$*. Five of these rewrite steps are useless for proving the considered embedding (the selected redex is underlined):*

$$+(2,\underline{+(3,1)}) \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} +(2,1) \qquad \underline{+(2,+(3,1))} \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} 1$$
$$+(2,\underline{+(3,1)}) \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} +(2,3) \qquad \underline{+(2,+(3,1))} \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} 2$$
$$\underline{+(2,+(3,1))} \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} +(3,1) \qquad \underline{+(2,+(3,1))} \to_{Emb((\Sigma^\sharp)^{\mathscr{U}})/B} 3$$

For a term with k *addends, we have* $(2^k)-2$ *rewriting steps. This leads to a huge combinatorial explosion when considering the complete rewrite search tree.*

In conclusion, there are three problems when trying to use Definition 4.13 to mechanize the order-sorted homeomorphic embedding relation modulo axioms $\mathrel{\breve{\trianglelefteq}}_B$ of Definition 4.10. First, the intrinsic non-determinism of the rules may unnecessarily produce an extremely large search space. Second, as shown in Example 4.8, this intrinsic non-determinism in the presence of axioms is intolerable, that is, unfeasible to handle. Third, the associated reachability problems do not scale up to complex embedding problems so that a suitable search strategy must be introduced. We address these problems stepwise in the sequel.

4.4 Goal-driven homeomorphic embedding modulo B

The formulation of homeomorphic embedding as a reachability problem by using the rewrite rules of Definition 4.13 generates a blind search that does not take advantage of the actual terms t and t' being compared for embedding. In this section, we provide a more refined formulation of homeomorphic embedding modulo axioms that is *goal driven* in the sense that, given an embedding problem for t and t', it inductively processes the terms t and t' in a top-down manner.

First, in the following section, we introduce a calculus that extends the homeomorphic embedding relation of Definition 4.5 to the order-sorted equational case.

4.4.1 An order-sorted homeomorphic embedding calculus modulo B

Let us introduce an order-sorted calculus for embeddability goals $t \overset{\vee}{\trianglelefteq}_B^{gd} t'$ that directly handles the algebraic axioms of B in the deduction system, with B being any combination of A and/or C axioms for the theory operators. Roughly speaking, this is achieved by specializing the Coupling rule of Definition 4.5 w.r.t. B. This calculus extends with sorts (kinds) a similar definition given in [Alpuente et al. 2017a], where kinds were not considered in the Variable inference rule of [Alpuente et al. 2017a].

Definition 4.15 (Goal-driven (order-sorted) homeomorphic embedding modulo B)**.** The homeomorphic embedding relation modulo B is defined as the smallest relation that satisfies the inference rules of Definition 4.5 together with the new inference rules given in Figure 4.5. These are:

1. the three inference rules (Variable, Diving, and Coupling) of Definition 4.5 for any function symbol;
2. one extra coupling rule for the case of a commutative symbol with or without associativity (Coupling_C);
3. two extra coupling rules for the case of an associative symbol with or without commutativity (Coupling_A); and
4. two extra coupling rules for the case of an associative-commutative symbol (Coupling_{AC}).

$$\textit{Coupling}_C \quad \frac{s_0 \overset{\vee}{\trianglelefteq}_B^{gd} t_1 \ \wedge \ s_1 \overset{\vee}{\trianglelefteq}_B^{gd} t_0}{f(s_0,s_1) \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,t_1)}$$

$$\textit{Coupling}_A \quad \frac{f(s_0,s_1) \overset{\vee}{\trianglelefteq}_B^{gd} t_0 \ \wedge \ s_2 \overset{\vee}{\trianglelefteq}_B^{gd} t_1}{f(s_0,f(s_1,s_2)) \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,t_1)} \qquad \frac{s_0 \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,t_1) \ \wedge \ s_1 \overset{\vee}{\trianglelefteq}_B^{gd} t_2}{f(s_0,s_1) \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,f(t_1,t_2))}$$

$$\textit{Coupling}_{AC} \quad \frac{f(s_0,s_1) \overset{\vee}{\trianglelefteq}_B^{gd} t_1 \ \wedge \ s_2 \overset{\vee}{\trianglelefteq}_B^{gd} t_0}{f(s_0,f(s_1,s_2)) \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,t_1)} \qquad \frac{s_1 \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,t_1) \ \wedge \ s_0 \overset{\vee}{\trianglelefteq}_B^{gd} t_2}{f(s_0,s_1) \overset{\vee}{\trianglelefteq}_B^{gd} f(t_0,f(t_1,t_2))}$$

FIGURE 4.5: Extra coupling rules for A, C, and AC symbols

Proposition 4.16. *Given Σ, B, and terms $t,t' \in \mathcal{T}_\Sigma(\mathcal{X})$, $t \breve{\trianglelefteq}_B t'$ iff $t^\sharp \breve{\trianglelefteq}^{gd}_B t'^\sharp$ where $t^\sharp, t'^\sharp \in \mathcal{T}_{\Sigma^\sharp}$.*

Proof. Immediate by considering that each inference rule of Figure 4.5 explicitly combines the Coupling inference rule with the corresponding A and/or C axioms. For example, the inference rule Coupling_C of Figure 4.5 is equivalent to the inference rule Coupling of Figure 4.3, but when the former considers an embedding problem $f(s_0,s_1)\breve{\trianglelefteq}^{gd}_B f(t_0,t_1)$, the latter considers both $f(s_1,s_0)\breve{\trianglelefteq}^{gd}_B f(t_0,t_1)$ and $f(s_0,s_1)\breve{\trianglelefteq}^{gd}_B f(t_1,t_0)$. □

Example 4.9. *Consider the binary symbol $+$ obeying associativity and commutativity axioms, and the terms $t = +(1,2)$ and $t' = +(2,+(3,1))$ of Example 4.8. We can prove $t \breve{\trianglelefteq}^{gd}_B t'$ by*

$$[\mathit{Coupling}_{AC}]\frac{[\mathit{Diving}]\dfrac{1\breve{\trianglelefteq}^{gd}_B 1}{1\breve{\trianglelefteq}^{gd}_B +(3,1)} \qquad 2\breve{\trianglelefteq}^{gd}_B 2}{+(1,2)\breve{\trianglelefteq}^{gd}_B +(2,+(3,1))}$$

We can also prove a more complex embedding goal by first using the right inference rule for AC of Figure 4.5 and then the generic Coupling and Diving inference rules.

$$[\mathit{Coupling}_{AC}]\frac{[\mathit{Coupling}_{AC}]\dfrac{[\mathit{Diving}]\dfrac{2\breve{\trianglelefteq}^{gd}_B 2}{2\breve{\trianglelefteq}^{gd}_B +(4,2)} \qquad 3\breve{\trianglelefteq}^{gd}_B 3}{+(2,3)\breve{\trianglelefteq}^{gd}_B +(+(4,2),3)} \qquad 1\breve{\trianglelefteq}^{gd}_B 1}{+(1,+(2,3))\breve{\trianglelefteq}^{gd}_B +(+(4,2),+(3,1))}$$

It is immediate to see that, when the size of the involved terms t and t' grows, the improvement in performance of $\breve{\trianglelefteq}^{gd}_B$ w.r.t. $\breve{\trianglelefteq}_B$ can be significant (just compare these two embedding proofs with the corresponding search trees for $\breve{\trianglelefteq}_B$).

4.4.2 Reachability-based, (order-sorted) goal-driven homeomorphic embedding formulation

Let us provide a more operational goal-driven characterization of the order-sorted homeomorphic embedding modulo B by means of the relation $\breve{\trianglelefteq}^{rbgd}_B$. We formalize it in the reachability style of Definition 4.13. The main challenge here is how to generate a suitable rewrite theory $R^{rbgd}(\Sigma,B)$ that can decide embedding modulo B by running a reachability goal.

Definition 4.17 (Goal-driven homeomorphic embedding rewrite rules modulo B)**.** Given Σ and B, a rewrite theory $R^{rbgd}(\Sigma,B) = (\Sigma,B,R)$ is defined as follows

1. For each particular instance of the inference rules of the form $\frac{}{u\breve{\trianglelefteq}^{gd}_B v}$ given in Definition 4.15 (e.g., the Variable Inference Rule from Definition 4.5 or the Coupling Inference Rule from Definition 4.5, for the case of a constant symbol c), we include in R a rewrite rule of the form $u\breve{\trianglelefteq}^{rbgd}_B v \to \mathit{true}$.
2. For each particular instance of the inference rules of the form $\frac{u_1\breve{\trianglelefteq}^{gd}_B v_1 \wedge \cdots \wedge u_k\breve{\trianglelefteq}^{gd}_B v_k}{u\breve{\trianglelefteq}^{gd}_B v}$ given in Definition 4.15, we include in R a rewrite rule of the form $u\breve{\trianglelefteq}^{rbgd}_B v \to u_1\breve{\trianglelefteq}^{rbgd}_B v_1 \wedge \cdots \wedge u_k\breve{\trianglelefteq}^{rbgd}_B v_k$.

Proposition 4.18. *Given Σ, B, and terms $t,t' \in \mathcal{T}_{\Sigma^\sharp}$, $t \breve{\trianglelefteq}^{gd}_B t'$ iff $(t\breve{\trianglelefteq}^{rbgd}_B t') \to^*_{R^{rbgd}(\Sigma^\sharp,B)/B} \mathit{true}$.*

Proof. Immediate by a straightforward transformation of inference rules into rewrite rules. □

Example 4.10. *Consider the binary symbol $+$ of Example 4.4. According to Definition 4.15 and the use of only ground terms, there are eleven inference rules for $\check{\trianglelefteq}_B^{gd}$:*

$$\textit{Diving}\qquad \frac{t \check{\trianglelefteq}_B^{gd} t'}{t \check{\trianglelefteq}_B^{gd} suc(t')} \qquad \frac{t \check{\trianglelefteq}_B^{gd} t_1}{t \check{\trianglelefteq}_B^{gd} +(t_1,t_2)} \qquad \frac{t \check{\trianglelefteq}_B^{gd} t_2}{t \check{\trianglelefteq}_B^{gd} +(t_1,t_2)}$$

$$\textit{Coupling}\qquad \frac{}{0 \check{\trianglelefteq}_B^{gd} 0} \qquad \frac{t \check{\trianglelefteq}_B^{gd} t'}{suc(t) \check{\trianglelefteq}_B^{gd} suc(t')} \qquad \frac{t_1 \check{\trianglelefteq}_B^{gd} t_1' \wedge t_2 \check{\trianglelefteq}_B^{gd} t_2'}{+(t_1,t_2) \check{\trianglelefteq}_B^{gd} +(t_1',t_2')}$$

$$\textit{Coupling}_C\qquad \frac{t_1 \check{\trianglelefteq}_B^{gd} t_2' \wedge t_2 \check{\trianglelefteq}_B^{gd} t_1'}{+(t_1,t_2) \check{\trianglelefteq}_B^{gd} +(t_1',t_2')}$$

$$\textit{Coupling}_A\qquad \frac{+(t_0,t_1) \check{\trianglelefteq}_B^{gd} t_1' \wedge t_2 \check{\trianglelefteq}_B^{gd} t_2'}{+(t_0,+(t_1,t_2)) \check{\trianglelefteq}_B^{gd} +(t_1',t_2')} \qquad \frac{t_1 \check{\trianglelefteq}_B^{gd} +(t_0',t_1') \wedge t_2 \check{\trianglelefteq}_B^{gd} t_2'}{+(t_1,t_2) \check{\trianglelefteq}_B^{gd} +(t_0',+(t_1',t_2'))}$$

$$\textit{Coupling}_{AC}\qquad \frac{+(t_0,t_1) \check{\trianglelefteq}_B^{gd} t_2' \wedge t_2 \check{\trianglelefteq}_B^{gd} t_1'}{+(t_0,+(t_1,t_2)) \check{\trianglelefteq}_B^{gd} +(t_1',t_2')} \qquad \frac{t_2 \check{\trianglelefteq}_B^{gd} +(t_0',t_1') \wedge t_1 \check{\trianglelefteq}_B^{gd} t_2'}{+(t_1,t_2) \check{\trianglelefteq}_B^{gd} +(t_0',+(t_1',t_2'))}$$

However, the corresponding TRS $R^{rbgd}(\Sigma,B)$ only contains seven rewrite rules because, due to pattern matching modulo associativity and commutativity in rewriting logic, the other rules are redundant:

$$\begin{array}{rrll}
\textit{(Diving)} & T & \check{\trianglelefteq}_B^{rbgd} suc(T') & \to T \check{\trianglelefteq}_B^{rbgd} T' \\
 & T & \check{\trianglelefteq}_B^{rbgd} +(T_1,T_2) & \to T \check{\trianglelefteq}_B^{rbgd} T_1 \\
\textit{(Coupling)} & \sharp & \check{\trianglelefteq}_B^{rbgd} \sharp & \to true \\
 & 0 & \check{\trianglelefteq}_B^{rbgd} 0 & \to true \\
 & suc(T) & \check{\trianglelefteq}_B^{rbgd} suc(T') & \to T \check{\trianglelefteq}_B^{rbgd} T' \\
\textit{(Coupling}_{\emptyset,C,A,AC}\textit{)} & +(T_1,T_2) & \check{\trianglelefteq}_B^{rbgd} +(T_1',T_2') & \to T_1 \check{\trianglelefteq}_B^{rbgd} T_1' \wedge T_2 \check{\trianglelefteq}_B^{rbgd} T_2'
\end{array}$$

For example, the rewrite sequence proving $+(1,+(2,3)) \check{\trianglelefteq}_B^{rbgd} +(+(4,2),+(3,1))$ is:

$$\begin{array}{l}
+(1,+(2,3)) \check{\trianglelefteq}_B^{rbgd} +(+(4,2),+(3,1)) \to_{R^{rbgd}(\Sigma^\sharp,B)/B} +(2,3)) \check{\trianglelefteq}_B^{rbgd} +(+(4,2),3) \wedge 1 \check{\trianglelefteq}_B^{rbgd} 1 \\
\qquad \to_{R^{rbgd}(\Sigma^\sharp,B)/B} 2 \check{\trianglelefteq}_B^{rbgd} +(4,2) \wedge 3 \check{\trianglelefteq}_B^{rbgd} 3 \\
\qquad \to_{R^{rbgd}(\Sigma^\sharp,B)/B} 2 \check{\trianglelefteq}_B^{rbgd} 2 \\
\qquad \to_{R^{rbgd}(\Sigma^\sharp,B)/B} true
\end{array}$$

Although the improvement in performance achieved by using the rewriting relation $\to_{R^{rbgd}(\Sigma^\sharp,B)/B}$ versus the rewriting relation $\to^*_{Emb(\Sigma^\sharp)/B}$ is important, the search space is still huge since the expression $+(1,+(2,3)) \check{\trianglelefteq}_B^{gd} +(+(4,2),+(3,1))$ matches the left-hand side $+(T_1,T_2) \check{\trianglelefteq}_B^{gd} +(T_1',T_2')$ in many different ways (e.g., $\{T_1 \mapsto 1, T_2 \mapsto +(2,3), \ldots\}$, $\{T_1 \mapsto 2, T_2 \mapsto +(1,3), \ldots\}$, $\{T_1 \mapsto 3, T_2 \mapsto +(1,2), \ldots\}$).

In the following section, we provide a more efficient calculus of homeomorphic embedding modulo axioms by considering equational (deterministic) normalization (thus avoiding search) and by exploiting the meta-level features of Maude (thus avoiding any theory generation).

4.5 Meta-Level deterministic (order-sorted) goal-driven homeomorphic embedding modulo B

The implementation of rewriting modulo equational axioms in high-performance languages such as Maude relies on a meta-level, flattened representation of terms that is rooted by poly-variadic versions of the associative (or associative-commutative) symbols of the term [Clavel et al., 2007, Chapter 14]. For instance, given an associative (or associative-commutative) symbol f with n arguments and $n \geq 2$, flattened terms rooted by f are canonical forms w.r.t. the set of rules given by the following rule schema

$$f(x_1,\ldots,f(t_1,\ldots,t_n),\ldots,x_m) \to f(x_1,\ldots,t_1,\ldots,t_n,\ldots,x_m) \quad n,m \geq 2$$

The flattened version of a term t is represented by $\underline{t}$. Given an associative (or associative-commutative) symbol f and a term $f(t_1,\ldots,t_n)$, those terms among the $t_1,\ldots,t_n$ that are not rooted by f are called *f-alien terms* (or simply *alien terms*). In the following, we implicitly consider that all terms are in canonical form.

Our first improvement for implementing the order-sorted homeomorphic embedding modulo equational axioms of Definition 4.15 is based on handling flattened terms explicitly.

Definition 4.19 (Flattened homeomorphic embedding modulo B)**.** The homeomorphic embedding relation modulo B, $\breve{\trianglelefteq}_B^{ml}$, *for terms in flattened form* is defined as the smallest relation that satisfies the following inference rules:

1. the three inference rules (Variable, Diving, and Coupling) of Definition 4.5 for any function symbol;
2. the extra coupling rule of Figure 4.5 for the case of a commutative symbol with or without associativity (Coupling_C);
3. the extra coupling rule of Figure 4.6 for the case of an associative symbol with or without commutativity (Coupling_A); and
4. the extra coupling rule of Figure 4.6 for the case of an associative-commutative symbol (Coupling_{AC}).

$$\textit{Coupling}_A \frac{\exists j \in \{1,\ldots,m-n+1\} : s_1 \breve{\trianglelefteq}_B^{ml} t_j \wedge f(s_2,\ldots,s_n) \breve{\trianglelefteq}_B^{ml} f(t_{j+1},\ldots,t_m) \wedge \forall k < j : s_1 \not\breve{\trianglelefteq}_B^{ml} t_k}{f(s_1,\ldots,s_n) \breve{\trianglelefteq}_B^{ml} f(t_1,\ldots,t_m)}$$

$$\textit{Coupling}_{AC} \frac{\exists j \in \{1,\ldots,m\} : s_1 \breve{\trianglelefteq}_B^{ml} t_j \wedge f(s_2,\ldots,s_n) \breve{\trianglelefteq}_B^{ml} f(t_1,\ldots,t_{j-1},t_{j+1},\ldots,t_m)}{f(s_1,\ldots,s_n) \breve{\trianglelefteq}_B^{ml} f(t_1,\ldots,t_m)}$$

FIGURE 4.6: Coupling rules for flattened terms with associative and associative-commutative symbols

Proposition 4.20. *Given Σ and B, for terms t and t' in $\mathscr{T}_\Sigma(\mathscr{X})$, $t \breve{\trianglelefteq}_B^{gd} t'$ iff $\underline{t} \breve{\trianglelefteq}_B^{ml} \underline{t}'$.*

Proof. Consider an embedding goal $t \breve{\trianglelefteq}_B^{gd} t'$ where both t and t' are rooted by an associate (resp. an associative-commutative) symbol f. Let us assume that the flattened versions are

$\underline{t} = f(t_1,\ldots,t_n)$ and $\underline{t}' = f(t'_1,\ldots,t'_m)$, with $m \geq n$. For the embedding goal $t \breve{\trianglelefteq}^{gd}_B t'$, the Coupling inference rule of Figure 4.1 and the Coupling$_A$ (resp. Coupling$_{AC}$) inference rule of Figure 4.5 can be applied to all of the elements in the equivalence class of t, i.e., the terms resulting from all possible rearrangements of t due to associativity (resp. associativity and commutativity). However, after several applications of the inference rules of $\breve{\trianglelefteq}^{gd}_B$, all of them end up with the form $t_i \breve{\trianglelefteq}^{gd}_B t'_j$ where $i \leq j$ (resp. $1 \leq i \leq n$ and $1 \leq j \leq m$). For the embedding goal $\underline{t} \breve{\trianglelefteq}^{ml}_B \underline{t}'$, the Coupling$_A$ (resp. Coupling$_{AC}$) inference rule of Figure 4.6 can be applied and after several applications of the inference rules, all of them end up with the same form. □

Flattened terms are only accessible in Maude through its META-LEVEL functional module. In the following, we provide a characterization of the order-sorted homeomorphic embedding relation $\breve{\trianglelefteq}^{ml}_B$ for flattened terms by (i) using a set of equations instead of rules and (ii) using meta-representations of terms instead of explicit terms of the given signature Σ. Rewriting logic is reflective [Clavel and Meseguer 2002], in the sense that some commands of Maude at the user level can be represented at the object level in a consistent way. In other words, the meta-level representation correctly simulates the relevant metatheoretic features of Maude such as loading a module, evaluating a term, or computing the least sort of a term. Reflection is systematically used in the design and implementation of the Maude language, making the metatheory of rewriting logic accessible to the user in a clear, principled, and efficient way. In the sequel, a variable x of sort `s` is meta-represented as $\bar{x} =$ `'`x`:s` and a non-variable term $t = f(t_1,\ldots,t_n)$, with $n \geq 0$, is meta-represented as $\bar{t} =$ `'`$f[\bar{t_1},\ldots,\bar{t_n}]$. See [Clavel et al., 2007, Chapter 14] for further details.

Definition 4.21 (Meta-level (order-sorted) homeomorphic embedding modulo B)**.** The meta-level (order-sorted) homeomorphic embedding modulo B, $\breve{\trianglelefteq}^{ml}_B$, of Definition 4.19 is defined for term meta-representations by means of the equational theory E^{ml} given in Figure 4.7, where the auxiliary meta-level functions **any** and **all** implement the existential and universal tests in the Diving and Coupling inference rules of Figure 4.1, and we introduce two new meta-level functions **all_A** and **all_AC** that implement existential tests of Figure 4.6, which are specific to A and AC symbols.

Example 4.11. *Given the embedding problem for terms* $+(1,+(2,3))$ *and* $+(+(4,2),+(3,1))$*, the corresponding call to the meta-level homeomorphic embedding* $\breve{\trianglelefteq}^{ml}_B$ *of Definition 4.21 is* `'+['1, '2, '3]` $\breve{\trianglelefteq}^{ml}_B$ `'+['4, '2, '3, '1]` *and its evaluation sequence is given in Figure 4.8, according to the equational theory* E^{ml} *of Figure 4.7.*

Proposition 4.22. *Given* Σ *and* B*, for terms* t *and* t' *in* $\mathcal{T}_\Sigma(\mathcal{X})$*,* $t \breve{\trianglelefteq}^{gd}_B t'$ *iff* $(\bar{t} \breve{\trianglelefteq}^{ml}_B \bar{t}')!_{\overrightarrow{E^{ml}/B}} = true$*.*

Proof. Immediate by realizing that the inference rules of Definition 4.19 can be easily transformed into a rewrite theory $R^{ml}(\Sigma,B)$, as we did in Definition 4.17 for the inference rules of Definition 4.15. This rewrite theory is terminating and can be easily transformed into a set of rules that are confluent, terminating, and coherent modulo associativity and commutativity axioms. And then those rewrite rules can be straightforwardly translated into the equational theory E^{ml} of Definition 4.21 which manipulates flattened terms at the meta-level. □

A further optimized version of Definition 4.21 can be obtained by replacing the Boolean conjunction (*and*) and disjunction (*or*) operators with the computationally more efficient Maude Boolean operators `and-then` and `or-else` which avoid evaluating the second argument when the result of evaluating the first one suffices to compute the result.

$$
\begin{array}{rll}
\sharp \trianglelefteq_B^{ml} \sharp & = \textbf{true} & \textit{for each } \sharp \textit{ in } \Pi_{\not\mathscr{K}} \\
F[TermList] \trianglelefteq_B^{ml} \sharp & = \textbf{false} & \\
T \trianglelefteq_B^{ml} F[TermList] & = \textbf{any}(T, TermList) & \text{if } root(T) \neq F \\
F[Term1] \trianglelefteq_B^{ml} F[Term2] & = Term1 \trianglelefteq_B^{ml} Term2 & \\
F[TermList1] \trianglelefteq_B^{ml} F[TermList2] & = \textbf{any}(F[TermList1], TermList2) & \\
& \quad \textbf{or} & \\
& \quad \textbf{all}(TermList1, TermList2) & \text{if } length(TermList_i) \neq 1, i \in \{1,2\} \\
F[U,V] \trianglelefteq_B^{ml} F[X,Y] & = \textbf{any}(F[U,V],[X,Y]) & \text{if } Ax(F) = \{C\} \\
& \quad \textbf{or}(\ U \trianglelefteq_B^{ml} X \textbf{ and } V \trianglelefteq_B^{ml} Y\ ) & \\
& \quad \textbf{or}\ (\ U \trianglelefteq_B^{ml} Y \textbf{ and } V \trianglelefteq_B^{ml} X\ ) & \\
F[TermList1] \trianglelefteq_B^{ml} F[TermList2] & = \textbf{any}(F[TermList1], TermList2) & \text{if } Ax(F) = \{A\} \\
& \quad \textbf{or} & \\
& \quad \textbf{all\_A}(TermList1, TermList2) & \\
F[TermList1] \trianglelefteq_B^{ml} F[TermList2] & = \textbf{any}(F[TermList1], TermList2) & \text{if } Ax(F) = \{A,C\} \\
& \quad \textbf{or} & \\
& \quad \textbf{all\_AC}(TermList1, TermList2) &
\end{array}
$$

$$
\begin{array}{rl}
\textbf{any}(U, nil) & = \textbf{false} \\
\textbf{any}(U, V:L) & = U \trianglelefteq_B^{ml} V \textbf{ or any}(U, L)
\end{array}
$$

$$
\begin{array}{rl}
\textbf{all}(nil, nil) & = \textbf{true} \\
\textbf{all}(nil, U:L) & = \textbf{false} \\
\textbf{all}(U:L, nil) & = \textbf{false} \\
\textbf{all}(U:L1, V:L2) & = U \trianglelefteq_B^{ml} V \textbf{ and all}(L1, L2)
\end{array}
$$

$$
\begin{array}{rl}
\textbf{all\_A}(nil, L) & = \textbf{true} \\
\textbf{all\_A}(U:L, nil) & = \textbf{false} \\
\textbf{all\_A}(U:L1, V:L2) & = (U \trianglelefteq_B^{ml} V \textbf{ and all\_A}(L1, L2)) \textbf{ or all\_A}(U:L1, L2))
\end{array}
$$

$$
\begin{array}{rl}
\textbf{all\_AC}(nil, L) & = \textbf{true} \\
\textbf{all\_AC}(U:L1, L2) & = \textbf{all\_AC\_Aux}(U:L1, L2, L2) \\
\textbf{all\_AC\_Aux}(U:L1, nil, L3) & = \textbf{false} \\
\textbf{all\_AC\_Aux}(U:L1, V:L2, L3) & = (U \trianglelefteq_B^{ml} V \textbf{ and all\_AC}(L1, \textbf{remove}(V, L3))) \\
& \quad \textbf{or} \\
& \quad \textbf{all\_AC\_Aux}(U:L1, L2, L3))
\end{array}
$$

$$
\begin{array}{rl}
\textbf{remove}(U, nil) & = \textbf{nil} \\
\textbf{remove}(U, V:L) & = \textbf{if } U = V \textbf{ then } L \textbf{ else } V : \textbf{remove}(U, L)
\end{array}
$$

FIGURE 4.7: Meta-level homeomorphic embedding modulo axioms

Definition 4.23 (Strategic meta-level deterministic embedding modulo B)**.** We define $\trianglelefteq_B^{sml}$ as the strategic version of relation $\trianglelefteq_B^{ml}$ that is obtained by replacing the Boolean operators `and` and `or` with Maude's `and-then` operator for the *short-circuit* version of the conjunction and the `or-else` operator for the short-circuit version of the disjunction [Clavel et al., 2007, Chapter 9.1], respectively.

The optimization idea behind Definition 4.23 was first proposed in [Alpuente et al. 2018]. In the following section, we propose two novel optimizations of order-sorted homeomorphic embedding modulo equational axioms B that can achieve further speedups.

$$
\begin{aligned}
\text{'}+[\text{'}1,\text{'}2,\text{'}3] \trianglelefteq^{ml}_{B} \text{'}+[\text{'}4,\text{'}2,\text{'}3,\text{'}1] &\rightarrow_{E^{ml}/B} \textbf{any}(\text{'}+[\text{'}1,\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\quad\ \textbf{or all\_AC}([\text{'}1,\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{false} \\
&\quad\ \textbf{or all\_AC}([\text{'}1,\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow_{E^{ml}/B} \textbf{all\_AC}([\text{'}1,\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],[\text{'}1],[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow_{E^{ml}/B} \text{'}1 \trianglelefteq^{ml}_{B} \text{'}1 \\
&\quad\ \textbf{and all\_AC}([\text{'}2,\text{'}3],\textbf{remove}(\text{'}1,[\text{'}4,\text{'}2,\text{'}3,\text{'}1])) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{all\_AC}([\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{all\_AC\_Aux}([\text{'}2,\text{'}3],[\text{'}2,\text{'}3],[\text{'}4,\text{'}2,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow_{E^{ml}/B} \text{'}2 \trianglelefteq^{ml}_{B} \text{'}2 \\
&\quad\ \textbf{and all\_AC}([\text{'}3],\textbf{remove}(\text{'}2,[\text{'}4,\text{'}2,\text{'}3])) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}2,\text{'}3],[\text{'}3],[\text{'}4,\text{'}2,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{all\_AC}([\text{'}3],[\text{'}4,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}2,\text{'}3],[\text{'}3],[\text{'}4,\text{'}2,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \text{'}3 \trianglelefteq^{ml}_{B} \text{'}3 \\
&\quad\ \textbf{and all\_AC}(\textbf{nil},\textbf{remove}(\text{'}3,[\text{'}4,\text{'}3])) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}2,\text{'}3],[\text{'}3],[\text{'}4,\text{'}2,\text{'}3]) \\
&\quad\ \textbf{or all\_AC\_Aux}([\text{'}1,\text{'}2,\text{'}3],\mathit{nil},[\text{'}4,\text{'}2,\text{'}3,\text{'}1]) \\
&\rightarrow^{*}_{E^{ml}/B} \textbf{true}
\end{aligned}
$$

FIGURE 4.8: Evaluation sequence for $\text{'}+[\text{'}1,\text{'}2,\text{'}3] \trianglelefteq^{ml}_{B} \text{'}+[\text{'}4,\text{'}2,\text{'}3,\text{'}1]$

4.6 Optimizations based on the term B-ordering and reachable kinds

Typically hidden inside the B-matching algorithms, some pertinent term transformations allow terms that contain operators obeying equational axioms to be rewritten into convenient B-normal forms that facilitate the matching modulo B. In the case of AC-theories, these transformations not only include translating the term to flattened form (as shown in Section 4.5) but also allowing terms to be reordered and suitably parenthesized in order to enable subsequent rewrite steps. Basically, this is achieved by producing a single, auxiliary representative of their AC congruence class (i.e., the AC-normal form). An AC-normal form is typically generated by replacing nested occurrences of the same AC operator by a flattened argument list under a variadic symbol, sorting these arguments under a fixed linear ordering and combining equal arguments using multiplicity superscripts [Eker 2003]. For example, the congruence class containing $f(f(\alpha,f(\beta,\alpha)),f(f(\gamma,\beta),\beta))$ where f is an AC symbol and subterms α, β, and γ belong to

alien theories might be represented by $f^*(\alpha^2, \beta^3, \gamma)$, where f^* is a variadic symbol that replaces nested occurrences of f. A more formal account of this transformation is given in [Eker 1995, 2003]. For any set B of axioms, Maude's total ordering on terms is written $\preceq_B$. This ordering is closed under context, i.e., $t \preceq_B t' \Rightarrow C[t] \preceq_B C[t']$ for any context C, and size-compatible, i.e., $t \preceq_B t' \Rightarrow size(t) \leq size(t')$ where $size(t)$ is the number of symbols in T.

Let us formulate a novel optimization $\check{\trianglelefteq}_B^{order+sml}$ of the order-sorted homeomorphic embedding modulo B that takes advantage of Maude's total order on terms $\preceq_B$. The optimization is based on adding a pruning equation that negatively concludes the test whenever $T' \prec_B T$ without actually running the call $T \check{\trianglelefteq}_B^{sml} T'$.

Definition 4.24 (Early pruning based on term order $\preceq_B$)**.** We extend the equational theory E^{ml} of Figure 4.7 (but with the strategic optimization given in Definition 4.23) with the following pruning equation that determines the (meta-level) embedding problem $T \check{\trianglelefteq}_B^{order+sml} T'$ negatively whenever $T' \prec_B T$.

$$T \check{\trianglelefteq}_B^{order+sml} T' = \textbf{if } T' \prec_B T \textbf{ then false else } T \check{\trianglelefteq}_B^{sml} T'$$

The correctness of the above optimization derives from the following two results.

Lemma 4.25. *If $T' \prec_B T$ then $T \check{\trianglelefteq}_B T'$ does not hold.*

Proof. If $T' \prec_B T$, then $T \neq T'$. Since $T' \prec_B T$ is equivalent to $T' \preceq_B T \wedge T \not\preceq_B T$ and $\preceq_B$ is size-compatible, then $T' \prec_B T$ implies $T \neq T'$ and $size(T') \leq size(T)$. Let us consider the cases when $size(T') = size(T)$ and $size(T') < size(T)$ separately. If $T \neq T'$ and $size(T') = size(T)$, then there is at least one clash position, i.e., a position in t and t' whose root symbol differ, hence the successive application of the coupling inference rule can never succeed, which implies $T \not\check{\trianglelefteq}_B T'$.

For the case when $size(T') < size(T)$, the proof follows directly from the following facts: (i) B is any combination of associativity and commutativity axioms for the binary function symbols in Σ; (ii) every associativity or commutativity axiom $l = r$ in B is size-preserving, i.e., $size(l) = size(r)$; and (iii) $\trianglelefteq$ is size-compatible, i.e., $T \trianglelefteq T'$ implies that $size(T) \leq size(T')$. From (i), (ii) and (iii) it follows that $\check{\trianglelefteq}_B$ is also size-compatible, i.e., $T \check{\trianglelefteq}_B T'$ implies that $size(T) \leq size(T')$. Therefore, $size(T') < size(T)$ implies that $T \not\check{\trianglelefteq}_B T'$ and the result follows. □

Example 4.12. *Consider the (flattened) terms $t = +(1,2,3)$ and $t' = +(0,1,2)$ for the signature of Example 4.4, and let T, T' be their corresponding meta-level representation. In Maude's total order for terms, $+(0,1,2) \prec_B +(1,2,3)$ (and correspondingly $T' \prec_B T$ as well). According to Definition 4.24, the embedding problem $T \check{\trianglelefteq}_B^{order+sml} T'$ immediately fails.*

Let us formalize a second optimization $\check{\trianglelefteq}_B^{kinds+sml}$ of $\check{\trianglelefteq}_B^{sml}$ that anticipates failure whenever there is a subterm of t whose kind is not in the set of all kinds that can be obtained (by instantiation) from the subterms of t'.

Definition 4.26 (Reachable kinds)**.** Given a term t, we let $kinds(t)$ denote the set of kinds of all of the subterms of t. Given the signature Σ, we compute $kinds(t)$ as the solution to a reachability problem using the following rewrite theory $Kinds(\Sigma) = (\Sigma', \emptyset, R)$, where Σ' contains the kinds of Σ as constants of a universal sort $\mathscr{U}$ and R consists of all rewrite rules

$$k \to k_i$$

where $(f : s_1, \ldots, s_n \to s)$ in Σ, $k = [s]$, $k_i = [s_i]$, $k \neq k_i$ and $i \in \{1, \ldots, n\}$. By using $\mathscr{R} = Kinds(\Sigma)$, we define $\mathbf{kinds}(t) = \{k \mid \lceil t \rceil \to^*_{\mathscr{R}} k\}$.

The intuition behind Definition 4.26 is that $kinds(t)$ represents the set of kinds of the subterms of any instance of t.

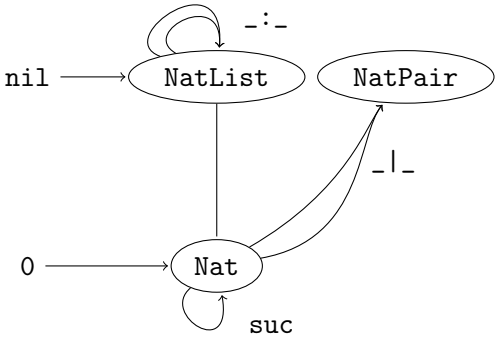


FIGURE 4.9: Signature graph of Example 4.13

Example 4.13. *Consider the following order-sorted signature* Σ *(graphically depicted in Figure 4.9), which extends the signature of natural numbers with two new sorts* `NatPair` *and* `NatList`*, which are aimed at representing pairs of natural numbers and sequences of natural numbers, respectively. Pairs are constructed by using the pair concatenation operator* `_|_`*, whereas lists are constructed by using* `nil` *and the (associative) list concatenation operator* `_:_`*. Note that there are two different kinds,* `[NatPair]` *and* `[NatList]`*, where* `[NatList]` *contains the sorts* `Nat` *and* `NatList` *because* `Nat < NatList`*.*

```
sorts Nat NatPair NatList .
subsort  Nat < NatList .
op 0 : -> Nat .
op suc : Nat -> Nat .
op _|_ : Nat Nat -> NatPair .
op nil -> NatList .
op _:_ : NatList NatList -> NatList [assoc] .
```

The rewrite theory $Kinds(\Sigma)$ *consists of:*

```
sort U .
ops [NatList] [NatPair] : -> U .
rl [NatPair] => [NatList] .
```

Now, given the term `Z:NatList`*, we have* $kinds($`Z:NatList`$) = \{$`[NatList]`$\}$*, whereas given the term* `X:Nat | Y:Nat`*, we have* $kinds($`X:Nat | Y:Nat`$) = \{$`[NatPair]`, `[NatList]`$\}$.

Definition 4.27 (Early prune based on reachable kinds)**.** We extend the equational theory E^{ml} of Figure 4.7 (with the strategic optimization given in Definition 4.23) with the following pruning

equation that decides the (meta-level) embedding problem $T \mathrel{\breve{\trianglelefteq}}_B^{kinds+sml} T'$ negatively whenever the set of all reachable kinds from T' does not contain all of the rechable kinds from T.

$$T \mathrel{\breve{\trianglelefteq}}_B^{kinds+sml} T' = \textbf{if } kinds(T) \not\subseteq kinds(T') \textbf{ then false else } T \mathrel{\breve{\trianglelefteq}}_B^{sml} T'$$

Example 4.14. *Consider again the order-sorted signature* Σ *for pairs and sequences of natural numbers of Example 4.13. The embedding goal* (`X:Nat | Y:Nat`) $\mathrel{\breve{\trianglelefteq}}_B$ `Z:NatList` *does not hold because the kind* `[NatPair]` *of the left term* (`X:Nat | Y:Nat`) *is not contained in the set* $kinds$(`Z:NatList`) *of reachable kinds from the right term, which only contains* `[NatList]`. *However, for the inverse goal we cannot conclude* `Z:NatList` $\not\mathrel{\breve{\trianglelefteq}}_B$ (`X:Nat | Y:Nat`) *because the set* $kinds$(`X:Nat | Y:Nat`) $= \{$`[NatPair]`, `[NatList]`$\}$ *does contain the only reachable kind* `[NatList]` *for the left term* `Z:NatList`. *And actually,* `Z:NatList` $\mathrel{\breve{\trianglelefteq}}_B$ (`X:Nat | Y:Nat`) *holds according to Definition 4.10 because* `Z:NatList` $\mathrel{\breve{\trianglelefteq}}_B$ `X:Nat`.

The correctness of the above optimization derives from the following result.

Lemma 4.28. *If* $kinds(T) \not\subseteq kinds(T')$*, then* $T \mathrel{\breve{\trianglelefteq}}_B T'$ *does not hold.*

Proof. (By contradiction). Assume that there is a subterm t of T such that $\lceil t \rceil \notin kinds(T')$ and that $T \mathrel{\breve{\trianglelefteq}}_B T'$ holds. Since $T \mathrel{\breve{\trianglelefteq}}_B T'$ then also $t \mathrel{\breve{\trianglelefteq}}_B T'$. Let us consider separately the case when t is ground and when t is non-ground. By Definition of $\mathrel{\breve{\trianglelefteq}}_B$, if t is ground, then every symbol of t must appear in T', hence it cannot happen that $\lceil t \rceil \notin kinds(T')$. If t is non-ground then every non-variable symbol of t must appear in T' and for each variable X of t there must be a variable Y in T' such that $\lceil X \rceil = \lceil Y \rceil$; hence, it cannot happen that $\lceil t \rceil \notin kinds(T')$. □

In the following section, we demonstrate that the two optimizations $\mathrel{\breve{\trianglelefteq}}_B^{order+sml}$ and $\mathrel{\breve{\trianglelefteq}}_B^{kinds+sml}$ are almost inexpensive and pay off in practice. We consider the two optimizations $\mathrel{\breve{\trianglelefteq}}_B^{order+sml}$ and $\mathrel{\breve{\trianglelefteq}}_B^{kinds+sml}$ separately as well as their natural combination in our last optimal order-sorted equation embedding relation $\mathrel{\breve{\trianglelefteq}}_B^{kosml}$ that (lazily) first checks the pruning condition based on reachable kinds of Definition 4.27 and then checks the pruning condition based on the term order $\prec_B$ of Definition 4.24. In other words,

$$\begin{aligned} T \mathrel{\breve{\trianglelefteq}}_B^{kosml} T' \quad = \; & \textbf{if } kinds(T) \not\subseteq kinds(T') \\ & \textbf{or-else} \\ & T' \prec_B T \\ & \textbf{then false} \\ & \textbf{else } T \mathrel{\breve{\trianglelefteq}}_B^{sml} T' \end{aligned}$$

Note that, we do not implement a recursive optimization that tries to verify the pruning conditions recursively within $\mathrel{\breve{\trianglelefteq}}_B^{sml}$. The reason is that the recursive checking of the pruning conditions would introduce a dramatic, exponential overhead so that the recursive optimization would prove ineffective.

4.7 Experiments

We develop HEMS, an equational homeomorphic embedding checker in Maude that implements all five equational homeomorphic embedding formulations $\mathrel{\breve{\trianglelefteq}}_B$, $\mathrel{\breve{\trianglelefteq}}_B^{rbgd}$, $\mathrel{\breve{\trianglelefteq}}_B^{ml}$, $\mathrel{\breve{\trianglelefteq}}_B^{sml}$, and $\mathrel{\breve{\trianglelefteq}}_B^{kosml}$ of

Benchmark	♯ Axioms				$\breve{\trianglelefteq}_B$			$\breve{\trianglelefteq}_B^{rbgd}$			$\breve{\trianglelefteq}_B^{ml}$, $\breve{\trianglelefteq}_B^{sml}$, $\breve{\trianglelefteq}_B^{kosml}$		
	∅	A	C	AC	♯E	♯R	GT(ms)	♯E	♯R	GT(ms)	♯E	♯R	GT(ms)
Kmp	9	0	0	0	0	15	1	0	57	2	21	0	0
NatList	5	1	1	2	0	10	1	0	26	1	21	0	0
Maze	5	1	0	1	0	36	7	0	787	15	21	0	0
Dekker	16	1	0	2	0	59	8	0	823	18	21	0	0

TABLE 4.1: Size of generated theories for naïve and goal-driven definitions vs. meta-level definitions

the previous sections. The HEMStool is publicly available at [HEMS Website]. The implementation consists of approximately 2.5K lines of Maude source code and is publicly available at [HEMS Website]. In this section, we provide an experimental comparison of the five equational homeomorphic embedding implementations by running a significant number of equational embedding goals. In order to compare the performance of the different implementations in the worst possible scenario, all benchmarked goals return false, which ensures that the whole search space for each goal has been completely explored, while the execution times for succeeding goals whimsically depend on the particular node of the search tree where success is found.

We tested our implementations on a 3.3GHz Intel Xeon E5-1660 with 64 GB of RAM running Maude v2.7.1, and we considered the average of ten executions for each test. We have chosen four representative programs: (i) *KMP*, the classical KMP string pattern matcher [Alpuente et al. 1998a]; (ii) *NatList*, a Maude implementation of lists of natural numbers; (iii) *Maze*, a non-deterministic Maude specification that defines a maze game in which multiple players must reach a given exit point by walking or jumping, where colliding players are eliminated from the game [Alpuente et al. 2015]; and (iv) *Dekker*, a Maude specification that models a faulty version of Dekker's protocol, one of the earliest solutions to the mutual exclusion problem that appeared in [Clavel et al. 2007]. As testing benchmarks, we considered a set of representative embeddability problems for the four programs that are generated during the execution of Maude's partial evaluator, Victoria [Alpuente et al. 2017a].

Tables 4.1, 4.2, 4.3, and 4.4 below analyze different aspects of the implementation. In Table 4.1, we compare the size of the generated rewrite theories for the naïve and the goal-driven definitions versus the meta-level definitions. For $\breve{\trianglelefteq}_B^{ml}$, $\breve{\trianglelefteq}_B^{sml}$, and $\breve{\trianglelefteq}_B^{kosml}$, there are the same number (21) of generated equations (♯E), whereas the number of generated rules (♯R) is zero because both definitions are purely equational (deterministic) and just differ in the version of the Boolean operators being used. As for the generated rewrite theories for computing $\breve{\trianglelefteq}_B$ and $\breve{\trianglelefteq}_B^{rbgd}$, they contain no equations, while the number of generated rules increases with the complexity of the program (that heavily depends on the equational axioms that the function symbols obey). The number of generated rules is much bigger for $\breve{\trianglelefteq}_B^{rbgd}$ than for $\breve{\trianglelefteq}_B$ (for instance, $\breve{\trianglelefteq}_B^{rbgd}$ is encoded by 823 rules for the Dekker program versus the 59 rules of $\breve{\trianglelefteq}_B$). Columns ∅, A, C, and AC summarize the number of free, associative, commutative, and associative-commutative symbols, respectively, for each benchmark program. The generation times (GT) are negligible for all rewrite theories.

To make the comparison on equal terms, in Tables 4.2 and 4.3, we compare the equational embedding implementations $\breve{\trianglelefteq}_B$, $\breve{\trianglelefteq}_B^{rbgd}$, $\breve{\trianglelefteq}_B^{ml}$, and $\breve{\trianglelefteq}_B^{sml}$ without adding the pruning conditions of Section 4.6 as they could be applicable to any of them. Our figures demonstrate that the implementation of $\breve{\trianglelefteq}_B^{sml}$ is the most efficient one among the four formulations. The performance improvement of the two optimizations of Section 4.6 w.r.t. $\breve{\trianglelefteq}_B^{sml}$ is evaluated in Table 4.4 below.

Benchmark	♯ Symbols		Size		$\check{\trianglelefteq}_B$	$\check{\trianglelefteq}_B^{rbgd}$	$\check{\trianglelefteq}_B^{ml}$	$\check{\trianglelefteq}_B^{sml}$
	A	AC	T1	T2	Time(ms)	Time(ms)	Time(ms)	Time(ms)
Kmp	0	0	5	5	10	6	1	1
				10	150	125	4	1
				100	TO	TO	280	95
				500	TO	TO	714	460
				1000	TO	TO	2184	1324
				5000	TO	TO	7528	5152
NatList	1	2	5	5	2508	2892	1	1
				10	840310	640540	1	1
				100	TO	TO	8	2
				500	TO	TO	60	5
				1000	TO	TO	102	10
				5000	TO	TO	210	85
Maze	1	1	5	5	40	25	1	1
				10	TO	20790	4	1
				100	TO	TO	256	2
				500	TO	TO	1980	10
				1000	TO	TO	10148	20
				5000	TO	TO	1057912	46
Dekker	1	1	5	5	50	40	1	1
				10	111468	110517	2	1
				100	TO	TO	5	3
				500	TO	TO	20	13
				1000	TO	TO	45	20
				5000	TO	TO	80	38

TABLE 4.2: Performance of equational homeomorphic embedding implementations w.r.t. problem size

For all benchmarks $T1 \trianglelefteq_B^{\alpha} T2$ in Table 4.2, we have fixed the size of $T1$, which is measured in the depth of (the non-flattened version of) the term, to five. As for $T2$, we have considered terms with increasing depths: 5, 10, 100, 500, 1000, and 5000. The ♯ `Symbols` column records the number of A (resp. AC) symbols occurring in the benchmarked goals.

The figures in Table 4.2 confirm our expectations regarding $\check{\trianglelefteq}_B$ and $\check{\trianglelefteq}_B^{rbgd}$ that the search space is huge and increases exponentially with the size of $T2$ (discussed for $\check{\trianglelefteq}_B$ in Example 4.8 and for $\check{\trianglelefteq}_B^{rbgd}$ in Example 4.9). Actually, when the size of $T2$ is 100 (and beyond), a given timeout (represented by TO in the tables) is reached that is set for 3.6e+6 milliseconds (1 h). The reader can also check that the more A,C, and AC symbols occur in the original program signature, the bigger the execution times. An odd exception is the Maze example, where the timeout is already reached for the size 10 of $T2$ even if the number of equational axioms is comparable to the other programs. This is because the AC-normalized, flattened version of the terms is much smaller than the original term size for the NatList and Dekker benchmarks but not for Maze, where the flattened and original terms have similar size. On the other hand, our experiments demonstrate that both $\check{\trianglelefteq}_B^{ml}$ and $\check{\trianglelefteq}_B^{sml}$ bring impressive speedups, with $\check{\trianglelefteq}_B^{sml}$ working outstandingly well in practice even for really complex terms where, taking into account the generous one hour time-out, the achieved speedup is at least 10^5 (and in Table 4.4 is further improved by one additional order of magnitude by enabling the optimizations of Section 4.6).

T1						T2						$\trianglelefteq_B$	$\trianglelefteq_B^{rbgd}$	$\trianglelefteq_B^{ml}$	$\trianglelefteq_B^{sml}$
Size		♯ Symbols				Size		♯ Symbols				Time(ms)	Time(ms)	Time(ms)	Time(ms)
OT	FT	∅	C	A	AC	OT	FT	∅	C	A	AC				
5	5	5	0	0	0	100	100	100	0	0	0	165	70	1	1
5	5	3	2	0	0	100	100	50	50	0	0	TO	TO	24	1
5	2	4	0	1	0	100	2	50	0	50	0	TO	TO	108035	3
5	2	4	0	0	1	100	2	50	0	0	50	TO	TO	42800	4
5	3	8	0	1	2	100	3	50	0	25	25	TO	TO	22796	5
5	5	5	0	0	0	500	500	500	0	0	0	48339	34000	12	4
5	5	3	2	0	0	500	500	250	250	0	0	TO	TO	1360	10
5	2	4	0	1	0	500	2	250	0	250	0	TO	TO	TO	30
5	2	4	0	0	1	500	2	250	0	0	250	TO	TO	TO	27
5	3	8	0	1	2	500	3	250	0	125	125	TO	TO	TO	50
5	5	5	0	0	0	1000	1000	1000	0	0	0	202224	88000	18	8
5	5	3	2	0	0	1000	1000	500	500	0	0	TO	TO	10184	40
5	2	4	0	1	0	1000	2	500	0	500	0	TO	TO	TO	50
5	2	4	0	0	1	1000	2	500	0	0	500	TO	TO	TO	56
5	3	8	0	1	2	1000	3	500	0	250	250	TO	TO	TO	87
5	5	5	0	0	0	5000	5000	1000	0	0	0	TO	TO	27	15
5	5	3	2	0	0	5000	5000	2500	2500	0	0	TO	TO	1159236	80
5	2	4	0	1	0	5000	2	2500	0	2500	0	TO	TO	TO	114
5	2	4	0	0	1	5000	2	2500	0	0	2500	TO	TO	TO	240
5	3	8	0	1	2	5000	3	2500	0	1250	1250	TO	TO	TO	368

TABLE 4.3: Performance of equational homeomorphic embedding implementations w.r.t. axiom entanglement for the NatList example

T1	T2	Case	$\trianglelefteq_B^{sml}$	$\trianglelefteq_B^{kinds+sml}$	$\trianglelefteq_B^{order+sml}$	$\trianglelefteq_B^{kosml}$
Size	Size		Time(ms)	Time(ms)	Time(ms)	Time(ms)
1000	1000	$kinds(T1) \not\subseteq kinds(T2)$ & $(T2 \not\lhd_B T1)$	12	2	15	2
5000	5000	$kinds(T1) \not\subseteq kinds(T2)$ & $(T2 \not\lhd_B T1)$	80	10	105	11
10000	10000	$kinds(T1) \not\subseteq kinds(T2)$ & $(T2 \not\lhd_B T1)$	240	15	270	18
50000	50000	$kinds(T1) \not\subseteq kinds(T2)$ & $(T2 \not\lhd_B T1)$	550	28	580	33
1000	1000	$kinds(T1) \subseteq kinds(T2)$ & $(T2 \lhd_B T1)$	10	12	4	6
5000	5000	$kinds(T1) \subseteq kinds(T2)$ & $(T2 \lhd_B T1)$	40	50	8	12
10000	10000	$kinds(T1) \subseteq kinds(T2)$ & $(T2 \lhd_B T1)$	140	160	14	20
50000	50000	$kinds(T1) \subseteq kinds(T2)$ & $(T2 \lhd_B T1)$	290	330	40	60

TABLE 4.4: Comparison of $\trianglelefteq_B^{sml}$ vs. the optimizations $\trianglelefteq_B^{kinds+sml}$, $\trianglelefteq_B^{order+sml}$, and $\trianglelefteq_B^{kosml}$

The reader may wonder how big is the impact of having A, C, or AC operators. In order to compare the relevance of these symbols, in Table 4.3 we fix one single benchmark program (NatList) that contains all three kinds of operators: two associative operators (list concatenation ; and natural division /), a commutative (natural pairing) operator (||), and two associative-commutative arithmetic operators $(+, *)$. With regard to the size of the considered terms, we compare the size of the original term (OT) with the size of its flattened version (FT); e.g., 500 versus 2 for the size of $T2$ in the last row. We have included the execution times of $\trianglelefteq_B$ and $\trianglelefteq_B^{rbgd}$ for completeness, but they do not reveal a dramatic improvement of $\trianglelefteq_B^{rbgd}$ with respect to $\trianglelefteq_B$

for the benchmarked (false) goals, contrary to what we initially expected. This means that $\trianglelefteq_B^{rbgd}$ cannot be generally used in real applications due to the risk of intolerable embedding test times, even if $\trianglelefteq_B^{rbgd}$ may be far less wasteful than $\trianglelefteq_B$ for succeeding goals, as discussed in Section 4.4. For $\trianglelefteq_B^{ml}$ and $\trianglelefteq_B^{sml}$, the figures show that the more A and AC operators comparatively occur in the problem, the bigger the improvement achieved. This is due to the following: (i) these two embedding definitions manipulate flattened meta-level terms; (ii) they are equationally defined, which has a much better performance in Maude than doing search; and (iii) our definitions are highly optimized for lists (that obey associativity) and sets (that obey both associativity and commutativity).

Homeomorphic embedding has been extensively used in Prolog for different purposes, such as termination analysis and partial deduction. In Figure 4.10, we have compared on a logarithmic scale our best embedding definition[4], $\trianglelefteq_B^{kosml}$, with a standard meta-level Prolog[5] implementation of the (syntactic) pure homeomorphic embedding $\trianglelefteq$ of Definition 4.5. We chose the NatList example and terms $T1$ and $T2$ which do not contain symbols obeying equational axioms as this is the only case that can be handled by the syntactic Prolog implementation. Our experiments show that our refined deterministic formulation $\trianglelefteq_B^{kosml}$ (i.e. without search) outperforms the Prolog version, so no penalty is incurred (actually, performance is increased) when syntactic embeddability tests are run in our equational implementation.

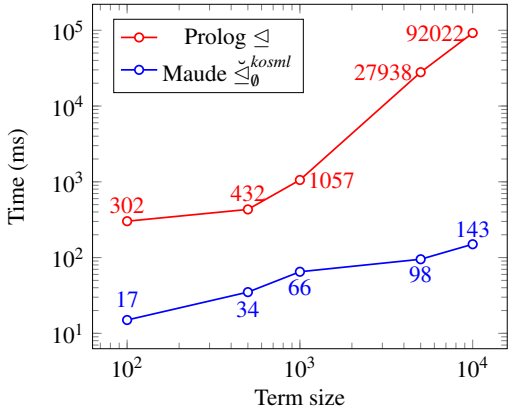


FIGURE 4.10: Comparison of $\trianglelefteq$ in Prolog vs. $\trianglelefteq_\emptyset^{kosml}$ for the NatList example (no axioms in goals)

Finally, Table 4.4 evaluates the two optimizations $\trianglelefteq_B^{order+sml}$ and $\trianglelefteq_B^{kinds+sml}$ separately as well as their combination with respect to our best implementation of order-sorted equational embedding, $\trianglelefteq_B^{kosml}$. In order to identify the optimization that achieves the best improvement, we have chosen eight pairs of terms $T1$ and $T2$ of (increasingly) equal size (1000, 5000, 10000, and 50000) and such that the embedding test fails because either (i) $kinds(T1) \not\subseteq kinds(T2)$ although $T2 \not\lhd_B T1$ or (ii) $kinds(T1) \subseteq kinds(T2)$ but $T2 \lhd_B T1$. In case (i), the optimization $\trianglelefteq_B^{kinds+sml}$ is enabled and the failure is anticipated without running $\trianglelefteq_B^{sml}$, whereas $\trianglelefteq_B^{order+sml}$ fruitlessly checks its pruning condition $T2 \lhd_B T1$ and then runs $\trianglelefteq_B^{sml}$ anyway. In case (ii), the optimization $\trianglelefteq_B^{kinds+sml}$ fruitlessly checks its pruning condition $kinds(T1) \not\subseteq kinds(T2)$ and then runs $\trianglelefteq_B^{sml}$ anyway, whereas the optimization $\trianglelefteq_B^{order+sml}$ is enabled and the failure is anticipated

[4]For this comparison, note that $\trianglelefteq_B^{kosml}$ boils down to $\trianglelefteq_B^{sml}$ since Prolog terms are unsorted and functors obey no axioms, hence no optimization based on sorts or term order is applicable.

[5]To avoid any bias, we took the Prolog code for the homeomorphic embedding of the ECCE system [Leuschel et al. 1998] that is available at `https://github.com/leuschel/ecce`, and we run it in SWI-Prolog 7.6.3.

without running $\check{\trianglelefteq}_B^{sml}$. Our figures reveal that the penalty that is incurred by checking the pruning condition of $\check{\trianglelefteq}_B^{order+sml}$ is slightly worse than for $\check{\trianglelefteq}_B^{kinds+sml}$, with the difference being more evident as terms grow (e.g., it takes twice as long for the term size of 5000: 580 ms in $\check{\trianglelefteq}_B^{order+sml}$ versus 330 ms in $\check{\trianglelefteq}_B^{kinds+sml}$). This justifies our final choice to first check $kinds(T1) \not\subseteq kinds(T2)$ and then eventually $T2 \prec_B T1$ in the definition of the optimal embedding relation formulation, $\check{\trianglelefteq}_B^{kosml}$, which achieves the best overall speedup.

Chapter 5

ACUOS^2: A High-performance System for Modular ACU Generalization with Subtyping and Inheritance

Computing generalizations is relevant in a wide spectrum of automated reasoning areas where analogical reasoning and inductive inference are needed, such as analogy making, case-based reasoning, web and data mining, ontology learning, machine learning, theorem proving, program derivation, and inductive logic programming, among others [Armengol 2007; Muggleton 1999; Ontañón and Plaza 2012].

In this chapter, we present ACUOS^2, a highly optimized implementation of the order-sorted ACU least general generalization algorithm formalized in [Alpuente et al. 2014b]. ACUOS^2 is a new, high-performance version of a previous prototype called ACUOS [Alpuente et al. 2014a]. ACUOS^2 runs up to five orders of magnitude faster than ACUOS and is able to solve complex generalization problems in which ACUOS fails to give a response. Both systems are written in Maude [Clavel et al. 2007], a programming language and system that implements rewriting logic [Meseguer 1992] and supports reasoning modulo algebraic properties, subtype polymorphism, and reflection. However, ACUOS was developed with a strong concern for simplicity and does not scale to real-life problem sizes, such as the biomedical domains often addressed in inductive logic programming and other AI applications, with a substantial number of variables, predicates and/or operators per problem instance. Scalability issues were not really unexpected since other equational problems (such as equational matching, equational unification, or equational embedding) are typically much more involved and costly than their corresponding "syntactic" counterparts, and achieving efficient implementations has required years of significant investigation effort.

In Section 5.1 we briefly summarize the problem of generalizing two (typed) expressions in theories that satisfy any combination of associativity (A), commutativity (C) and unity axioms (U). Section 5.2 explains the main functionality of the ACUOS^2 system and describes the novel implementation ideas and optimizations that have boosted the tool performance. A nontrivial application of equational generalization to a biological domain is described in Section 5.3. An in-depth experimental evaluation of ACUOS^2 is given in Section 5.4. Finally, in Section 5.5 we briefly discuss some related work.

5.1 Least General Generalization modulo A, C, and U

Computing a *least general generalization* (lgg) for two expressions t_1 and t_2 means finding the least general expression t such that both t_1 and t_2 are instances of t under appropriate substitutions. For instance, the expression `olympics(X,Y)` is a generalizer of both `olympics(1900,paris)` and `olympics(2024,paris)` but their least general generalizer, also known as *most specific generalizer* (msg) and *least common anti-instance* (lcai), is `olympics(X,paris)`.

Syntactic generalization has two important limitations. First, it cannot generalize common data structures such as records, lists, trees, or (multi-)sets, which satisfy specific premises such as, e.g., the order among the elements in a set being irrelevant. Second, it does not cope with types and subtypes, which can lead to more specific generalizers.

Consider the predicates `connected`, `flights`, `visited`, and `alliance` among cities, and let us introduce the constants `rome`, `paris`, `nyc`, `bonn`, `oslo`, `rio`, and `ulm`. Assume that the predicate `connected` is used to state that a pair of cities `C1;C2` are connected by transportation, with ";" being the *unordered pair constructor* operator so that the expressions `connected(nyc;paris)` and `connected(paris;nyc)` are considered to be equivalent modulo the commutativity of ";". The expressions `connected(nyc;paris)` and `connected(paris;bonn)` can be generalized to `connected(C;paris)`, whereas the syntactic least general (or most specific) generalizer of these two expressions is `connected(C1;C2)`.

Similarly, assume that the predicate `flights(C,L)` is used to state that the city `C` has direct flights to all of the cities in the list `L`. The *list concatenation* operator "`.`" records the cities[1] in the order given by the travel distance from `C`. Due to the associativity of list concatenation, i.e.,`(X.Y).Z = X.(Y.Z)`, we can use the flattened list `rio.paris.oslo.nyc` as a very compact and convenient representation of the congruence class (modulo associativity) whose members are the different parenthesized list expressions `((rio.paris).oslo).nyc`, `(rio.(paris.oslo)).nyc`, `rio.(paris.(oslo.nyc))`, *etc.* Then, for the expressions `flights(rome,paris.oslo.nyc.rio)` and `flights(bonn,ulm.oslo.rome)`, the least general generalizer is `flights(C,L1.oslo.L2)`, which reveals that `oslo` is the only common city that has a direct flight from `rome` and `bonn`. Note that `flights(C,L1.oslo.L2)` is more general (modulo associativity) than `flights(rome,paris.oslo.nyc.rio)` by the substitution {`C/rome, L1/paris, L2/(nyc.rio)`} and more general than `flights(bonn,ulm.oslo.rome)` by the substitution {`C/bonn, L1/ulm, L2/rome`}.

Due to the equational axioms ACU, in general there can be more than one least general generalizer of two expressions. As a simple example, let us record the travel history of a person using a list that is ordered by the chronology in which the visits were made; e.g., `visited(paris.paris.bonn.nyc)` denotes that `paris` has been visited twice before visiting `bonn` and then `nyc`. The travel histories `visited(paris.paris.bonn.nyc)` and `visited(bonn.bonn.rome)` have two incomparable least general generalizers: (a) `visited(L1.bonn.L2)` and (b) `visited(C.C.L)`, meaning that (a) the two travelers visited `bonn`, and (b) they consecutively repeated a visit to their own first visited city. Note that the two generalizers are least general and incomparable, since neither of them is an instance (modulo associativity) of the other.

Furthermore, consider the predicate `alliance(S)` that checks whether the cities in the set `S` have established an alliance. We introduce a new operator "`&`" that satisfies associativity, commutativity, and unit element `∅`; i.e., `X & ∅ = X` and `∅ & X = X`. We can use the flattened,

[1] A single city is automatically coerced into a singleton list.

multi-set notation `alliance(nyc & oslo & paris & rome)` as a very compact and convenient representation (with a total order on elements given by the lexicographic order) for the congruence class modulo ACU whose members are all of the different parenthesized permutations of the considered cities. Such permutations include as many occurrences of $\emptyset$ as needed, due to unity [Clavel et al. 2007]. In this scenario, the expressions (i) `alliance(nyc & oslo & paris & rome)` and (ii) `alliance(bonn & paris & rio & rome)` have an infinite set of ACU generalizers of the form `alliance(paris & rome & S1 & ⋯ & Sn)` yet they are all equivalent modulo ACU-renaming[2] so that we can choose one of them, typically the smallest one, as the class representative.

Regarding the handling of types and subtypes, let us assume that the constants `rome`, `paris`, `oslo`, `ulm`, and `bonn` belong to type European and that `nyc` and `rio` belong to type American. Furthermore, let us suppose that European and American are subtypes of a common type City that, in turn, is a subtype of the type Cities that can be used to model the typed version of the previous ACU (multi-)set structure. Subtyping implies automatic coercion: for instance, a European city also belongs to the type City and Cities. Note that the empty set, denoted by the unity $\emptyset$, only belongs to Cities.

In this typed environment, the above expressions (i) and (ii) have only one typed ACU least general generalizer `alliance(paris & rome & C1:American & C2:European)` that we choose as the representative of its infinite ACU congruence class. Note that `alliance(paris & rome & S:Cities)` is not a least general generalizer since it is strictly more general; it suffices to see that the typed ACU-lgg above is an instance of it modulo ACU with substitution $\{$`S:Cities/(C1:American & C2:European)`$\}$.

For a discussion on how to achieve higher-order generalization in Maude we refer to [Alpuente et al. 2014a].

5.2 ACUOS²: A High Performance Generalization System

ACUOS² is a new, totally redesigned implementation of the ACUOS system presented in [Alpuente et al. 2014a] that provides a remarkably faster and more optimized computation of least general generalizations. Generalizers are computed in an order-sorted, typed environment where inheritance and subtype relations are supported modulo any combination of associativity, commutativity, and unity axioms.

Both ACUOS and ACUOS² implement the generalization calculus of [Alpuente et al. 2014b] but with remarkable differences concerning how they deal with the combinatorial explosion of different alternative possibilities; see [Pottier 1989] for some theoretical results on the complexity of generalization. Consider the generalization problem

$$\texttt{connected(paris;bonn)} \stackrel{\triangle}{=} \texttt{connected(bonn;paris)}$$

that is written using the syntax of [Alpuente et al. 2014b]. ACUOS already includes some optimizations but follows [Alpuente et al. 2014b] straightforwardly and decomposes this problem (modulo commutativity of "`;`") into two simpler subproblems:

$$(P_1)\ \texttt{paris} \stackrel{\triangle}{=} \texttt{bonn} \wedge \texttt{bonn} \stackrel{\triangle}{=} \texttt{paris} \qquad (P_2)\ \texttt{paris} \stackrel{\triangle}{=} \texttt{paris} \wedge \texttt{bonn} \stackrel{\triangle}{=} \texttt{bonn}$$

[2] i.e., the equivalence relation $\approx_{ACU}$ induced by the relative generality (subsumption) preorder $\leq_{ACU}$, i.e., $s \approx_{ACU} t$ iff $s \leq_{ACU} t$ and $t \leq_{ACU} s$.

According to [Alpuente et al. 2014b], both are explored non-deterministically even if only the last subproblem would lead to the least general generalization. Much worse, due to axioms and types, a post-generation, time-expensive filtering phase is necessary to get rid of non-minimal generalizers. We have derived four groups of optimizations: (a) avoid non-deterministic exploration; (b) reduce the number of subproblems; (c) prune non-minimal paths to anticipate failure; and (d) filter out non-minimal solutions more efficiently.

(a) While ACUOS directly encoded the inference rules of [Alpuente et al. 2014b] as rewrite rules that non-deterministically compute generalizers by exploring all branches of the search tree in a don't-know manner, i.e., each branch potentially leads to a different solution, ACUOS² smartly avoids non-deterministic exploration by using *synchronous rewriting* [Baader and Nipkow 1998], also called *maximal parallel rewriting*, that allows ACUOS² to keep all current subproblems in a single data structure, e.g. $P_1 \mid P_2 \mid \cdots \mid P_n$, where all subproblems are simultaneously executed, avoiding any non-deterministic exploration at all. Synchronous rewriting is achieved in Maude by reformulating rewrite rules as oriented equations and, thanks to the different treatment of rules and equations in Maude [Clavel et al. 2007], the deterministic encoding of the inference rules significantly reduces execution time and memory consumption. Also, built-in Maude *memoization* techniques are applied to speed up the evaluation of common subproblems, which can appear several times during the generalization process.

(b) Enumeration of all possible terms in a congruence class is extremely inefficient, and even nonterminating when the U axiom is considered. Therefore, it should not be used to effectively solve generalization problems when A, AC, or ACU axiom combinations are involved. For instance, if f is AC, the term $f(a_1, f(a_2, \ldots, f(a_{n-1}, a_n), \ldots))$ has $(2n-2)!/(n-1)!$ equivalent combinations; this number may grow exponentially for generalization problems that contain several symbols obeying distinct combinations of axioms.

ACUOS² avoids class element enumeration (specifically the expensive computation of argument permutations for *AC* operators). Instead, it relies on the extremely efficient Maude built-in support for equational matching to decompose generalization problems into simpler subproblems, thereby achieving a dramatic improvement in performance.

(c) It is extremely convenient to discard as early as possible any generalization subproblem that will not lead to a least general generalization. For example, trivial generalization problems such as `paris` $\stackrel{\triangle}{=}$ `paris` are immediately solved once and for all without any further synchronous rewrite. Similarly, dummy generalization problems with single variable generalizers such as `nyc` $\stackrel{\triangle}{=}$ `paris` are solved immediately. However, note that `paris.oslo` $\stackrel{\triangle}{=}$ `nyc.oslo` is not a dummy problem. ACUOS² also checks whether a subproblem is more general than another during the whole process, discarding the more general one. For instance, P_1 above contains two dummy subproblems and P_2 above contains two trivial subproblems, which *safely* allows ACUOS² to discard P_1 as being more general than P_2.

(d) Getting rid of non-minimal generalizers commonly implies too many pairwise comparisons, i.e., whether a generalizer l_1 is an instance *modulo* axioms of a generalizer l_2, or vice versa. Term size is a very convenient ally here since a term t' being bigger than another term t prevents t from being an instance of t'. Note that this property is no longer true when there is a unit element. For instance, `alliance(nyc & rome & S1:Cities & S2:Cities)` is bigger (modulo ACU) than `alliance(nyc & rome & S:Cities)`; but

the latter is an instance of the former by the substitution {`S1/S, S2/0`}. Term size can reduce the number of matching comparisons by half.

The ACUOS² backend has been implemented in Maude and consists of about 2300 lines of code. It can be directly invoked in the Maude environment by calling the generalization routine `lggs(M,t1,t2)`, which facilitates ACUOS² being integrated with third-party software. Furthermore, ACUOS² functionality can be accessed through an intuitive web interface that is publicly available at [ACUOS² Website].

5.3 ACU Generalization in a Biological Domain

In this section, we show how ACUOS² can be used to analyze biological systems, e.g., to extract similarities and pinpoint discrepancies between two cell models that express distinct cellular states. We consider cell states that appear in the MAPK (Mitogen-Activated Protein Kinase) metabolic pathway that regulates growth, survival, proliferation, and differentiation of mammalian cells. Our cell formalization is inspired by and slightly modifies the data structures used in Pathway Logic (PL) [Talcott 2008] —a symbolic approach to the modeling and analysis of biological systems that is implemented in Maude. Specifically, a cell state can be specified as a typed term as follows.

We use sorts to classify cell entities. The main sorts are `Chemical`, `Protein`, and `Complex`, which are all subsorts of sort `Thing`, which specifies a generic entity. Cellular compartments are identified by sort `Location`, while `Modification` is a sort that is used to identify post-transactional protein modifications, which are defined by the operator "`[-]`" (e.g., the term `[EgfR - act]` represents the Egf (epidermal growth factor) receptor in an active state). A complex is a compound element that is specified by means of the associative ad commutative (AC) operator "`<=>`", which combines generic entities together.

A *cell state* is represented by a term of the form `[cellType | locs]`, where `cellType` specifies the cell type[3] and `locs` is a list (i.e., an associative data structure whose constructor symbol is "`,`") of cellular compartments (or locations). Each location is modeled by a term of the form `{ locName | comp }`, where `locName` is a name identifying the location (e.g., `CLm` represents the cell membrane location), and `comp` is a soup (i.e., an associative and commutative data structure with unity element `empty`) that specifies the entities included in that location. Note that cell states are built by means of a combination of A, AC, and ACU operators. The full formalization of the cell model is as follows.

```
fmod CELL-STRUCTURE is
  sorts Protein Thing Complex Chemical .
  subsorts Protein Complex Chemical < Thing .
  op _<=>_ : Thing Thing -> Complex [assoc comm] .
  ops Egf EgfR Pi3k Gab1 Grb2 Hras Plcg Sos1 Src : -> Protein .
  ops PIP2 PIP3 : -> Chemical .

  sort Soup .
  subsort Thing < Soup .
  op empty : -> Soup .
```

[3] For simplicity, we only consider mammalian cells denoted by the constant `mcell`.

```
  op __ : Soup Soup -> Soup [assoc comm id: empty] .

  sort Modification .
  ops act GTP GDP : -> Modification .
  op [_-_] : Protein   Modification -> Protein   .

  sort Location LocName Locations .
  subsort Location < Locations .
  op {_|_} : LocName Soup -> Location .
  ops CLc CLm CLi : -> LocName .
  op _,_ : Locations Locations -> Locations [assoc] .

  sorts Cell CellType .
  op [_|_] : CellType Locations -> Cell .
  op mcell : -> CellType .
endfm
```

Example 5.1. *The term c_1*

```
[ mcell | { Clc | Gab1 Grb2 Plcg Sos1 },
          { CLm | EgfR PIP2           },
          { CLi | [Hras - GDP] Src    } ]
```

models a cell state of the MAPK pathway with three locations: the cytoplasm (`CLc`*) includes five proteins* `Gab1`, `Grb2`, `Pi3k`, `Plcg`, *and* `Sos1`*; the membrane (*`CLm`*) includes the protein* `EgfR` *and the chemical* `PIP2`*; the membrane interior (*`CLi`*) includes the proteins* `Hras` *(modified by* `GDP`*) and* `Src`.

In this scenario, ACUOS² can be used to compare two cell states, c_1 and c_2. Indeed, any ACUOS² solution is a term whose non-variable part represents the common cell structure shared by c_1 and c_2, while its variables highlight discrepancy points where the two cell states differ.

Example 5.2. *Consider the problem of generalizing the cell state of Example 5.1 plus the following MAPK cell state c_2*

```
[ mcell | { CLc | Gab1 Plcg Sos1            },
          { CLm | PIP2 Egf <=> [EgfR - act] },
          { CLi | Grb2 Src [Hras - GDP]     } ]
```

For instance, ACUOS² *computes (in 4ms) the following least general generalizer*

```
[ mcell | { CLc | Gab1 Plcg Sos1 X1:Soup   },
          { CLm | PIP2 X2:Thing            },
          { CLi | Src X3:Soup [Hras - GDP] } ]
```

where `X1:Soup`, `X2:Thing`, *and* `X3:Soup` *are typed variables. Each variable in the computed lgg detects a discrepancy between the two cell states. The variable* `X2:Thing` *represents a generic entity that abstracts two distinct elements in the membrane location* `CLm` *of the two cell states. In fact, c_1's membrane includes the (inactive) receptor* `EgfR`*, whereas c_2's membrane*

contains the complex `Egf <=> [EgfR - act]` *that activates the receptor* `EgfR` *and binds it to the ligand* `Egf` *to start the metabolic process. Variables* `X1:Soup` *and* `X3:Soup` *indicate a protein relocation for* `Grb2`*, which appears in the location* `CLc` *in* c_1 *and in the membrane interior* `CLi` *in* c_2*. Note that the computed sort* `Soup` *is key in modeling the absence of* `Grb2` *in a location, since it allows* `X1:Soup` *and* `X3:Soup` *to be bound to the* `empty` *soup.*

5.4 Experimental Evaluation

To empirically evaluate the performance of ACUOS² we have considered the same generalization problems that were used to benchmark ACUOS in [Alpuente et al. 2014a], together with some additional problems that deal with complex ACU structures such as graphs and biological models. All of the problems are available online at the tool web site [ACUOS² Website] where the reader can also reproduce all of the experiments we conducted through the ACUOS² web interface. Specifically, the benchmarks used for the analysis are: (i) `incompatible types`, a problem without any generalizers; (ii) `twins`, `ancestors`, `spouses`, `siblings`, and `children`, some problems borrowed from the logic programming domain which are described in [Alpuente et al. 2014a]; (iii) `only-U`, a generalization problem modulo (just) unity axioms, i.e., without A and C; (iv) `synthetic`, an involved example mixing A, C, and U axioms for different symbols; (v) `multiple inheritance`, which uses a classic example of multiple subtyping from [Clavel et al. 2007] to illustrate the interaction of advanced type hierarchies with order-sorted generalization; (vi) `rutherford`, the classical analogy-making example that recognizes the duality between Rutherford's atom model and the solar system [Gentner 1983]; (vii) `chemical`, a variant of the case-based reasoning problem for chemical compounds discussed in [Armengol 2007]; (viii) `alliance`, the ACU example of Section [5.1]; (ix) `graph`, the leading example of [Baumgartner et al. 2018]; and (x) `biological`, the cell model discussed in Section [5.3].

We tested our implementations on a 3.30 GHz Intel(R) Xeon(R) E5-1660 with 64Gb of RAM memory running Maude v2.7.1, and we considered the average of ten executions for each test. Table 5.1 shows our experimental results. For each problem, we show the size (i.e., number of operators) of the input terms, the computation time (ms.) until the first generalization is found[4], and the number $\hat{S}$ of different subproblems that were generated so far, as a measure of how much the complexity of the problem has been simplified (before the optimizations, the number of produced subproblems was typically in the thousands for a term size of 100). In many cases, we cannot compare the time taken by each system to compute the set of all lggs, since the previous prototype ACUOS times out (for a generous timeout that we set to 60 minutes). Indeed, when we increase the size of the input terms from 20 to 100, the generalization process in ACUOS stops for most of the benchmarks due to timeout.

Considering the high combinatorial complexity of the ACU generalization problem, our implementation is highly efficient. All of the examples discussed in [Alpuente et al. 2014a], except for `incompatible types`, `twins (C)`, and `synthetic (C + AU)`, fail to produce a generalization in ACUOS when the problem size is 100, whereas the time taken in ACUOS² is in the range from 1 to 11067ms ($\sim$11s). In all of the benchmarks, our figures demonstrate an impressive performance boost w.r.t. [Alpuente et al. 2014a]: a speed up of five orders of magnitude for all of the ACU benchmarks.

[4]The computation time for the `incompatible types` benchmark is the same for any input term since we provide two input terms of incompatible sorts.

Benchmark	$\sharp S$	**Size**	ACUOS	ACUOS²	**Speedup**
			T1(ms)	T2(ms)	× (T1/T2)
incompatible types	0	20	30	1	30
	0	100	30	1	30
twins (C)	16	20	70	8	9
	42	100	23934	70	340
ancestors (A)	10	20	48	1	48
	31	100	TO	48	$>10^5$
spouses (A)	10	20	49	1	49
	31	100	TO	50	$>10^5$
spouses (AU)	10	20	531747	5	$\sim 10^5$
	61	100	TO	30	$>10^5$
siblings (AC)	16	20	TO	1	$>10^5$
	23	100	TO	150	$>10^5$
children (ACU)	12	20	TO	2	$>10^5$
	29	100	TO	3451	$>10^5$
only-U (U)	9	20	24	2	12
	9	100	TO	630	$>10^5$
synthetic (C+AU)	5	20	55	1	55
	5	100	31916	50	638
multiple inheritance (AC)	17	20	TO	10	$>10^5$
	31	100	TO	11067	$>10^5$
rutherford (AC+A+C)	5	20	48	1	48
	42	100	TO	320	$>10^5$
chemical (AU)	15	20	112	1	112
	31	100	TO	10	$>10^5$
graph (ACU+AU)	11	20	TO	1	$>10^5$
	31	100	TO	1002	$>10^5$
biological (ACU+AC+A)	22	20	TO	4	$>10^5$
	71	100	TO	50	$>10^5$
alliance (ACU)	11	20	TO	1	$>10^5$
	31	100	TO	9159	$>10^5$

TABLE 5.1: Experimental results

5.5 Related work

Related (but essentially different) problems of anti-unification for feature terms have been studied by [Aït-Kaci and Sasaki 2001], [Armengol 2007], and [Armengol and Plaza 2006]. The minimal and complete unranked anti-unification of [Baumgartner et al. 2013] and the term graph anti-unification of [Baumgartner et al. 2018] (together with the commutative extension) are also related to our work. The unranked anti-unification problems of [Baumgartner et al. 2013, 2018] can be directly solved by using our techniques for associative anti-unification with the unit element by simply introducing sorts to distinguish between term variables and hedge variables (and their instantiations) [Baumgartner et al. 2013]. Conversely, it is possible to simulate our calculus for associative least general generalization with the unit element in the minimal and complete unranked anti-unification algorithm of [Baumgartner et al. 2013], but not the rules for associative-commutative least general generalization with the unit element.

As for the generalization of feature terms, this problem has two main similarities with computing (least general) generalizations modulo combinations of A, C, and U axioms: 1) feature terms are order-sorted (in contrast to the unsorted setting of unranked term anti-unification); and 2) there is no fixed order for arguments. However, the capability to deal with recursive, possibly cyclic data structures such as graphs in ACU anti-unification does not seem to have

its counterpart in feature term anti-unification. Moreover, to generalize theories with a different number of clauses/equations (or a different number of atoms per clause), feature generalization algorithms resort to *ad hoc* mechanisms such as *background theories* and *projections* [Gentner 1983], whereas our approach naturally handles these kinds of generalizations by defining operators that obey the unity axiom.

Chapter 6

A Partial Evaluation Framework for Order-sorted Equational Programs modulo Axioms

This chapter focuses on the foundations of our order-sorted equational partial evaluation system, i.e., the core notions, principles, and algorithms. To the best of our knowledge, this is the first partial evaluation framework in the literature for order-sorted equational theories that is able to cope with subsorts, subsort polymorphism, convergent rules (equations), and equational axioms. We base our partial evaluator, Victoria, on a suitably generalized version of the general NPE procedure of [Alpuente et al. 1998a], which is parametric w.r.t. the *unfolding rule* used to construct finite computation trees and also w.r.t. an *abstraction operator* that is used to guarantee that only finitely many expressions are evaluated. For unfolding, we use *folding variant narrowing* [Escobar et al. 2012], which is an optimal narrowing strategy for convergent equational theories that computes *most general variants* modulo algebraic axioms and is efficiently implemented in Maude. For the abstraction, we rely on the *order-sorted equational least general generalization* recently investigated in [Alpuente et al. 2014b, 2019b]. As in [Alpuente et al. 1996], we follow the on-line approach to PE that makes control decisions about specialization on the fly, which is simpler to describe and offers better opportunities for powerful automated strategies than off-line partial evaluation [Christensen and Glück 2004; Jones et al. 1993], where decisions are made before specialization by using abstract data descriptions that are represented as program annotations. Nevertheless, we are able to tune the power of the specialization by distinguishing two groups of equations: the set of equations $\vec{E}$ that are used for rewriting modulo B and the set of equations $\vec{G} \subseteq \vec{E}$ that are only used for narrowing modulo B, where both $(\Sigma, B, \vec{E})$ and $(\Sigma, B, \vec{G})$ are convergent.

We introduced the idea in the paper [Alpuente et al. 2017a], but we did not present a fully automated tool where both a Maude equational program and an initial call were given and the tool would return the specialized program. In contrast, in this chapter we present a fully automated mature system that scales up to bigger and more complex specialization problems.

This chapter is organized as follows. In Section 6.1 we briefly discuss some related work. Section 6.2 formalizes our partial evaluation scheme by generalizing to the order-sorted and modulo axioms setting the key ingredients of NPE, namely unfolding (based on narrowing), closedness, homeomorphic embedding, and abstraction (based on generalization). Section 6.3 presents some experiments with the partial evaluator Victoria that implements our technique

which demonstrate the usefulness of our approach, with some specialized programs running up to two orders of magnitude (100 times) faster than the original one.

6.1 Related work

Program specialization has been investigated within different programming paradigms and applied to a wide variety of languages. Among the vast literature on program specialization, the partial evaluation of functional logic programs is the closest to our work. For the (rewriting-based) functional logic language Escher, a partial evaluator was described in [Lafave and Gallagher 1998]. The work by Darlington and Pull [Darlington et al. 1991] is the most clear predecessor of narrowing-driven partial evaluation. They proposed the use of narrowing as an alternative to the combination of instantiation and unfolding—in the sense of Burstall and Darlington [Burstall and Darlington 1977]—to perform partial evaluation. Their approach yielded a partial evaluator for the functional language HOPE extended with unification. For the functional logic language Curry [Hanus 2013], a partial evaluator based on needed narrowing was first proposed in [Albert et al. 1999]. The most recent partial evaluator for functional logic programs is described in [Hanus and Peemöller 2014], which is able to deal with Curry programs that may contain non–deterministic operations. For a recent discussion regarding the practical partial evaluation of mainstream languages such as JavaScript, Ruby, and R, see [Würthinger et al. 2017]. Obviously, none of these PE systems can deal with the salient features (subtype polymorphism and computing modulo axioms) considered in this work.

6.2 Specializing Equational Theories modulo Axioms

In this section, we introduce a partial evaluation algorithm for the decomposition $(\Sigma, B, \vec{E})$ of an equational theory $(\Sigma, \mathscr{E})$, with $\mathscr{E} = E \uplus B$, that is based on computing *folding variant narrowing* trees, and we establish the correctness of the transformation system. Our partial evaluation algorithm extends the general NPE procedure of [Alpuente et al. 1998a], which is parametric w.r.t. an unfolding rule to construct finite derivations for an expression and an abstraction operator used to guarantee that only finitely many expressions are evaluated.

This section is organized as follows. In Section 6.2.1, we recall the key ideas of the NPE approach. In Section 6.2.2, we discuss how the specialization of programs that contain sorts, subsorts, rules, and equational axioms is significantly more elaborate. In Section 6.2.3, we present the general algorithm for order sorted equational partial evaluation modulo axioms based on folding variant narrowing. Local termination of the general algorithm is discussed in Section 6.2.4, whereas global termination is discussed in Section 6.2.5. In Section 6.2.6, a post-processing algorithm is presented that gets rid of unnecessary symbols and further optimizes the program. Finally, in Section 6.2.7, we illustrate our equational NPE framework by specializing the interpreter of an imperative programming languages w.r.t. some input configurations.

6.2.1 The NPE Approach

Let us illustrate the classical NPE method with the following example that shows its ability to perform *deforestation* [Wadler 1990], a popular transformation that neither standard partial evaluation nor partial deduction can achieve [Alpuente et al. 1998a]. Essentially, the aim of

deforestation is to eliminate useless intermediate data structures, thus reducing the number of passes over data.

Example 6.1. *Consider the following Maude program that computes the mirror image of a (non-empty) binary tree, which is built with the free constructor* `_{_}_` *that stores an element as root above two given (sub-)trees, its left and right children. Note that this is a simple specialization problem, where the considered program does not contain any equational attributes either for* `_{_}_` *or for the operation* `flip` *defined therein:*

```
fmod FLIP-TREE is
  protecting NAT .
  sort NatTree .
  subsort Nat < NatTree .
  var N : Nat .
  vars R L : NatTree .
  op _{_}_ : NatTree Nat NatTree -> NatTree .
  op flip : NatTree -> NatTree .
  eq flip(N) = N [variant] .
  eq flip(L {N} R) = flip(R) {N} flip(L) [variant] .
endfm
```

By executing in Maude the input term `flip(flip(T))`*, this program returns the original tree* `T` *back, but it first computes an intermediate, mirrored tree* `flip(T)` *of* `T`*, which is then flipped again.*

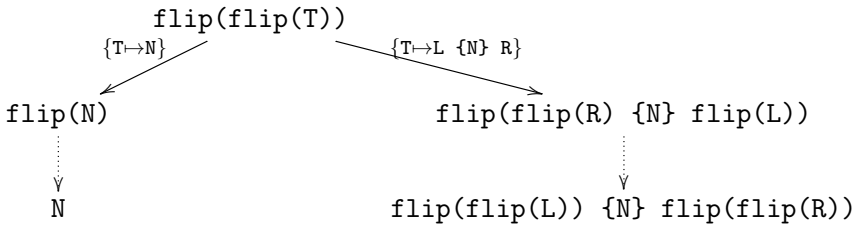


FIGURE 6.1: Folding variant narrowing tree for the goal `flip(flip(T))`.

Let us partially evaluate the input term `flip(flip(T))` *following the NPE approach. We compute the folding variant narrowing tree depicted*[1] *in Figure 6.1. This tree does not contain, altogether, uncovered calls in its leaves. Thus, after introducing the new symbol* `dflip`*, we get the following residual program:*

```
eq dflip(N) = N .
eq dflip(L {N} R) = dflip(L) {N} dflip(R) .
```

which is completely deforested, since the intermediate tree constructed after the first application of `flip` *is not constructed in the residual program using the specialized definition of* `dflip`*. This is equivalent to the program generated by deforestation [Wadler 1990] but with a much better*[2] *performance (see Section 6.3). Note that the fact that folding variant narrowing [Escobar et al. 2012] ensures normalization of terms at each step is essential for computing the calls*

[1] We show narrowing steps in solid arrows and rewriting steps in dotted arrows.

[2] Similarly to [Wadler 1990], the optimal program `dflip(T) = T` cannot be produced by our equational NPE technique.

`flip(flip(R))` *and* `flip(flip(L))` *that appear in the rightmost leaf of the tree in Figure 6.1, which are closed w.r.t. the root node of the tree.*

When we specialize programs that contain sorts, subsorts, rules, and equational axioms, things get considerably more involved, as discussed in the following section.

6.2.2 Partial evaluation of convergent rules modulo axioms

Let us motivate the problem by considering the following variant of the `flip` function of Example 6.1 for (binary) graphs instead of trees.

Example 6.2. *Consider the following Maude program for flipping binary graphs whose nodes may contain explicit, left and right references (pointers) to their child nodes in the graph. We use symbol* `#` *to denote an empty pointer. The* `BinGraph` *constructor* `_;_` *obeys associativity, commutativity, and identity (ACU) axioms so that it can be seen as a multiset of nodes* `{R1 I R2}`*, with* `R1` *and* `R2` *being references and* `I` *the node identifier. We provide for an unbounded number of (natural) node identifiers by establishing the subsort relation* `Nat < Id` .

```
fmod GRAPH is
  sorts BinGraph Node Id Ref .
  subsort Node < BinGraph .
  subsort Id < Ref .
  op {___} : Ref Id Ref -> Node .
  op mt : -> BinGraph .
  op _;_ : BinGraph BinGraph -> BinGraph [assoc comm id: mt] .
  op # : -> Ref . --- Void pointer
  ops 0 1 2 3 4 : -> Id .
  var I : Id .
  vars R1 R2 : Ref .
  var BG : BinGraph .
endfm
```

We are interested in flipping a graph and define a function `flip` *that takes a binary graph and returns the flipped graph.*

```
op flip : BinGraph -> BinGraph .
eq [E1] : flip(mt) = mt  [variant] .
eq [E2] : flip({R1 I R2} ; BG) = {R2 I R1} ; flip(BG) [variant] .
```

We can represent the graph shown on the left-hand side of Figure 6.2 as the following term `g` *of sort* `BinGraph`*:*

```
{1 0 2} ; {# 1 #} ; {3 2 4} ; {# 3 4} ; {# 4 0}
```

By invoking `flip(g)`*, the graph shown on the right-hand side of Figure 6.2 is computed.*

In order to specialize the previous program for the call `flip(flip(BG))`, we need several PE ingredients that have to be generalized to the corresponding (order–sorted) *equational* notions: (i) *equational closedness*, (ii) *equational embedding*, and (iii) *equational generalization*. These notions are discussed in the following sections.

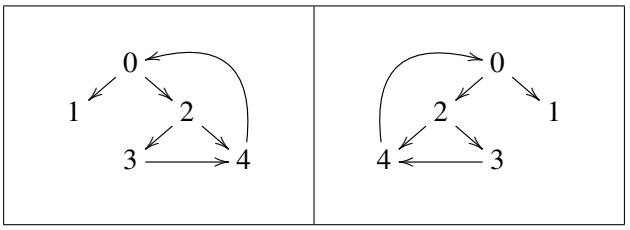


FIGURE 6.2: A binary graph (left) and its flipped version (right).

6.2.3 Equational closedness and the generalized Partial Evaluation scheme

In order to extend the NPE approach to equational theories $(\Sigma, B, \vec{E})$, we need to start by constructing a finite (possibly partial) $(\vec{E}, B)$-narrowing tree for the input call t using the folding variant narrowing strategy [Escobar et al. 2012], and then extracting the specialized rules $t\sigma \Rightarrow r$ (resultants) for each narrowing derivation $t \leadsto^{+}_{\sigma,\vec{E},B} r$ in the tree. In order to guarantee that all possible executions for t in the original program $(\Sigma, B, \vec{E})$ are covered by the specialization, we need to formalize an extended notion of closedness ensuring that any $(\vec{E}, B)$-narrowable subterm in the leaves of the tree can also be narrowed modulo B in the specialized rules. This ensures that resultants form a complete description covering all calls that may occur at run-time.

Let us define a general notion of *equational closedness* which relies on subsumption modulo B for theories whose function symbols (both defined and constructor symbols) can obey a set B of equational axioms.

Definition 6.1 (Equational Closedness)**.** Let $(\Sigma, B, \vec{E})$ be an equational theory decomposition and Q be a finite set of Σ-terms, i.e., terms that are built from Σ and a countably infinite set of variables $\mathscr{X}$. Assume the signature Σ splits into a set $\mathscr{D}$ of defined function symbols and a set $\mathscr{C}$ of constructor symbols, so that $\Sigma = \mathscr{D} \uplus \mathscr{C}$. We say that a Σ-term t is closed *modulo* B (w.r.t. Q and Σ), or simply B–closed, if $closed_B(Q,t)$ holds, where the predicate $closed_B$ is defined as follows:

$$closed_B(Q,t) \Leftrightarrow \begin{cases} true & \text{if } t \in \mathscr{X} \\ closed_B(Q,t_1) \wedge \ldots \wedge closed_B(Q,t_n) & \text{if } t = c(\overline{t_n}),\ c \in \mathscr{C},\ n \geq 0 \\ \bigwedge_{x \mapsto t' \in \theta} closed_B(Q,t') & \text{if } \exists q \in Q, \exists \theta \text{ such that} \\ & root(t) = root(q) \in \mathscr{D} \text{ and} \\ & q\theta =_B t \end{cases}$$

A set T of terms is closed modulo B (w.r.t. Q and Σ) if $closed_B(Q,t)$ holds for each t in T. A set R of rules is closed modulo B (w.r.t. Q and Σ) if the set that can be formed by taking the right-hand sides of all of the rules in R also is closed modulo B. We often omit Σ when no confusion can arise.

Example 6.3. *In order to partially evaluate the program in Example 6.2 w.r.t. the input term* `flip(flip(BG))`*, we set* $Q = \{$`flip(flip(BG))`$\}$ *and start by constructing the folding variant narrowing tree that is shown*[3] *in Figure 6.3.*

[3]To ease reading, the arcs of the narrowing tree are labelled with the corresponding equation applied at each narrowing step.

When we consider the leaves of the tree, we identify two requirements for Q-closedness, with B being ACU: (i) closed$_B(Q,t_1)$ with $t_1 =$ `mt` *and (ii) closed$_B(Q,t_2)$ with $t_2 =$* `{R1 I R2} ; flip(flip(BG'))`*. The call closed$_B(Q,t_1)$ holds straightforwardly (i.e., it is reduced to true) since the* `mt` *leaf is a constant and cannot be narrowed. The call closed$_B(Q,t_2)$ also returns true because* `{R1 I R2}` *is a flat constructor term and* `flip(flip(BG'))` *is a (syntactic) renaming of the root of the tree.*

We now show an example that requires using B-matching in order to ensure equational closedness modulo B.

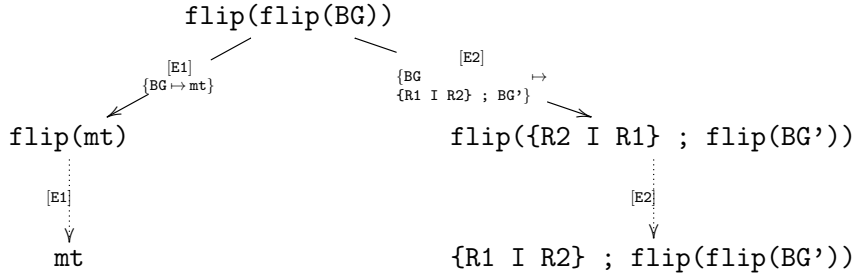


FIGURE 6.3: Folding variant narrowing tree for the goal `flip(flip(BG))`.

Example 6.4. *Let us introduce a new sort* `BinGraph?` *to encode bogus graphs that may contain spurious nodes in a supersort* `Id?` *and homomorphically extend the rest of symbols and sorts. For simplicity, we just consider one additional constant symbol* `e` *of sort* `Id?`*.*

```
sorts BinGraph? Id? Node? Ref? .
subsorts BinGraph Node? < BinGraph? .
subsort Node < Node? .
subsort Id < Id? .
subsorts Ref Id? < Ref? .
op e : -> Id? .
op {___} : Ref? Id? Ref? -> Node? .
op _;_ : BinGraph? BinGraph? -> BinGraph? [assoc comm id: mt] .
vars I I1 : Id .    var I? : Id? .
vars R1 R2 : Ref .  vars R1? R2? : Ref? .
var BG : BinGraph . var BG? : BinGraph? .
```

Let us consider a function `fix` *that receives an extended graph* `BG?`*, an unwanted node* `I?`*, and a new content* `I`*, and that traverses the graph replacing* `I?` *by* `I`*.*

```
op fix : Id Id? BinGraph? -> BinGraph .
eq [E3] : fix(I, I?, {R1? I? R2?} ; BG?) =
          fix(I, I?, {R1? I R2?} ; BG?) [variant] .
eq [E4] : fix(I, I?, {I? I1 R2?} ; BG?) =
          fix(I, I?, {I I1 R2?} ; BG?) [variant] .
eq [E5] : fix(I, I?, {R1? I1 I?} ; BG?) =
          fix(I, I?, {R1? I1 I} ; BG?) [variant] .
eq [E6] : fix(I, I?, BG) = BG [variant] .
```

For example, consider the following term t of sort `BinGraph?`*:*

```
{# 1 e} ; {e 0 #} ; {e e 3} ; {e 3 #}
```

that represents the graph shown on the left-hand side of Figure 6.4. By invoking `fix(2, e, t)`*, we can fix the graph t, by computing the corresponding transformed graph shown on the right-hand side of Figure 6.4, where the unwanted node e has been replaced.*

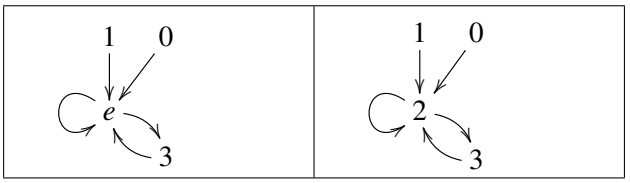


FIGURE 6.4: Fixing a graph.

Now assume we want to specialize the above function `fix` *w.r.t. the input term* `fix(2, e, {R1 I R2} ; BG?)`*, that is, a bogus graph with at least one non-spurious node* `{R1 I R2}` *(it is non-spurious because of the sort of variable* `I`*). Following the proposed methodology, we set* $Q =$ `{fix(2, e, {R1 I R2} ; BG?)}` *and start by constructing the folding variant narrowing tree shown in Figure 6.5.*

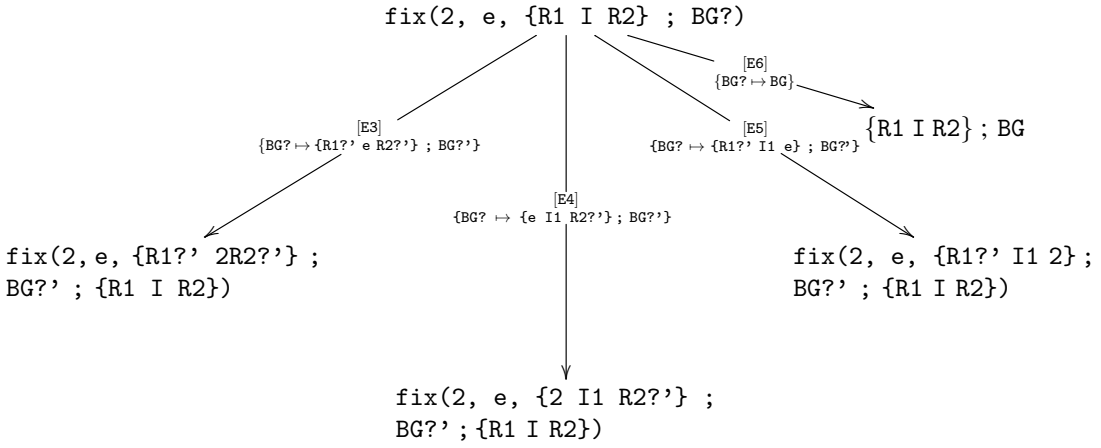


FIGURE 6.5: Folding variant narrowing tree for the goal `fix(2, e, {R1 I R2} ; BG?)`.

The right leaf `{R1 I R2} ; BG` *is a constructor term and cannot be unfolded. The first two branches to the left of the tree are closed modulo ACU with the root of the tree in Figure 6.5. For instance, for the left leaf*

$$t = \texttt{fix(2, e, \{R1?' 2 R2?'\} ; BG?' ; \{R1 I R2\})}$$

the condition $closed_B(Q,t)$ *is reduced*[4] *to true because t is an instance (modulo ACU) of the root node of the tree, and the subterm* $t' =$ `({R1?' 2 R2?'} ; BG?')` *occurring in the corresponding ACU-matcher is a constructor term. The other branches can be proved ACU-closed with the tree root in a similar way.*

[4]Note that this is only true because pattern matching modulo *ACU* is used for checking closedness.

Example 6.5 (Example 6.4 continued)**.** *Now let us assume that the function* `flip` *of Example 6.1 is replaced by the following definition extended to (bogus graphs of sort)* `BinGraph?`*, where the former equation* `E2` *is an instance of the new equation* `E2a`*:*

```
op flip : BinGraph? -> BinGraph? .
eq [E1x] : flip(mt) = mt  [variant] .
eq [E2a] : flip({R1? I  R2?} ; BG?) = {R2? I  R1?} ; flip(BG?) [variant] .
eq [E2b] : flip({R1? I? R2?} ; BG?) = {R2? I? R1?} ; flip(BG?) [variant] .
```

We specialize the whole program containing functions `flip` *and* `fix` *w.r.t. input term* `flip(fix(2, e, flip(BG)))`*; that is, take a graph* `BG`*, flip it, then fix any occurrence of nodes* `e`*, and finally flip it again. The corresponding folding variant narrowing tree is shown in Figure 6.6. Unfortunately this tree does not represent all possible computations for (any ACU-instances of) the input term, since the narrowable redexes occurring in the tree leaves are not a recursive instance of the only partially evaluated call so far,* `flip(fix(2, e, flip(BG)))`*. That is, the term*

```
flip(fix(2, e, flip(BG') ; {R2 I R1}))
```

in the rightmost leaf is not ACU-closed w.r.t. the root node of the tree. As in NPE, we need to introduce a methodology that recurses (modulo B) over the structure of the terms to augment the set of specialized calls in a controlled way, so as to ensure that all possible calls are covered by the specialized program.

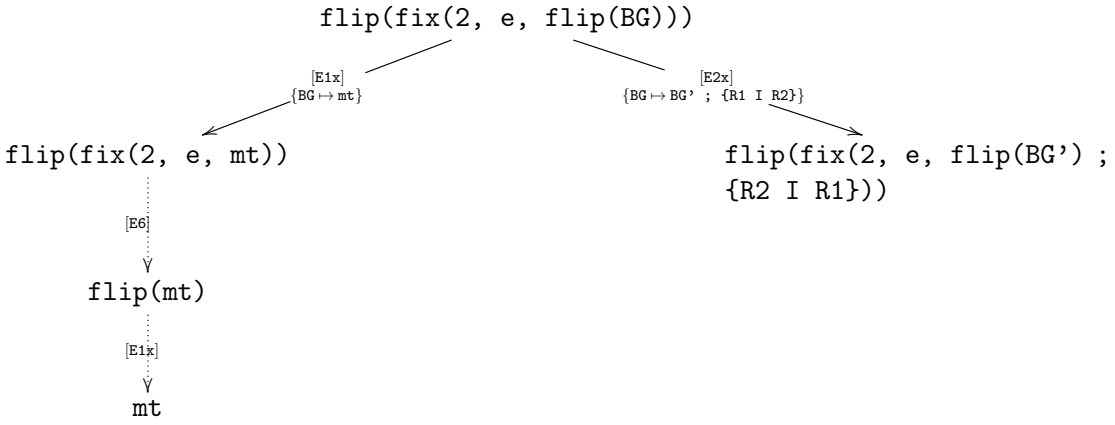


FIGURE 6.6: Folding variant narrowing tree for the goal `flip(fix(2, e, flip(BG)))`.

We are now ready to formulate the backbone of our partial evaluation methodology for equational theories that crystallize the ideas of the example above. We define a generic algorithm (Algorithm 1) that is parameterized by:

1. a *narrowing relation* (with narrowing strategy $\mathscr{S}$) that constructs search trees,
2. an *unfolding rule* that determines when and how to terminate the construction of the trees, and
3. an *abstraction operator* that is used to guarantee that the set of terms obtained during partial evaluation (i.e., the set of deployed narrowing trees) is kept finite.

Algorithm 1 Partial Evaluation for Equational Theories

Require:
An order-sorted rewrite theory $\mathscr{R} = (\Sigma, B, \vec{E})$ and a set of terms Q to be specialized in $\mathscr{R}$
Ensure:
A set Q' of terms s.t. UNFOLD$(Q', \mathscr{R}, \mathscr{S})$ is *closed* modulo B w.r.t. Q'

```
1: function EQNPE(R,Q,S)
2:     Q := Q↓_{E,B}
3:     repeat
4:         Q' := Q
5:         L ← UNFOLD(Q', R, S)
6:         Q ← ABSTRACT(Q', L, B)
7:     until Q' =_B Q
8:     return Q'
```

Note that, by using the notion of decomposition, the partial evaluation of an equational theory $(\Sigma, E \uplus B)$ can be seen as a particular case of this parameterized algorithm that is defined for rewrite theories $(\Sigma, B, \vec{E})$, and we prefer to keep it generic for further instantiations of this algorithm to deal with more complex rewrite theories.

Informally, the algorithm proceeds as follows. Given the input theory $\mathscr{R}$ and the set of terms Q, the first step consists in applying the unfolding rule UNFOLD$(Q, \mathscr{R}, \mathscr{S})$ to compute a finite (possibly partial) narrowing tree in $\mathscr{R}$ for each term t in Q and return the set $\mathscr{L}$ of the (normalized) leaves of the tree. Then, instead of proceeding directly with the partial evaluation of the terms in $\mathscr{L}$, an abstraction operator ABSTRACT$(Q, \mathscr{L}, B)$ is applied that properly combines each uncovered term in $\mathscr{L}$ with the (already partially evaluated) terms of Q so that the infinite growing of Q is avoided. The abstraction phase yields a new set of terms which may need further specialization, and, thus, the process is iteratively repeated while new terms are introduced.

Note that Algorithm 1 does not explicitly compute a partially evaluated theory $\mathscr{R}' = (\Sigma, B, \vec{E}')$. It does so implicitly, by computing the set Q' of partially evaluated terms (that unambiguously determine $\vec{E}'$ as the set $R_{Q', \mathscr{R}}$ of resultants $t\sigma \Rightarrow r$ associated to the root-to-leaf derivations $t \leadsto^{+}_{\sigma, \vec{E}, B} r$ in the tree, with t in Q'), such that the closedness condition for $\vec{E}'$ modulo B w.r.t. Q' is satisfied.

For the correctness of Algorithm 1, we require any instance of the generic abstraction operator ABSTRACT$(Q, \mathscr{L}, B)$ to agree with the following definition.

Definition 6.2 (Equational Abstraction)**.** Given the finite set of terms T and the already evaluated set of terms Q, ABSTRACT(Q, T, B) returns a new set Q' such that:

1. if $v \in Q'$, then there exists $u \in (Q \cup T)$ such that $u_{|p} =_B v\theta$ for some position p and substitution θ, and
2. for all $t \in (Q \cup T)$, t is closed with respect to Q' modulo B.

Roughly speaking, condition (1) ensures that the abstraction operator does not "create" new function symbols (i.e., symbols not present in the input arguments), whereas condition (2) ensures that the resulting set of terms "covers" (modulo B) the calls previously specialized and that equational closedness is preserved throughout successive abstractions.

There are two correctness issues for a PE procedure: *termination*, i.e., given any input goal, execution should always reach a stage at which there is no way to continue; and (partial) *correctness*, i.e., the residual program behaves as the original one for the considered input terms (provided PE terminates). The basic correctness of the transformation is ensured whenever $\mathscr{R}'$ is closed modulo B w.r.t. Q', i.e., every call in (the right-hand side of the rules in) $\mathscr{R}'$ is a (recursive) instance (modulo B) of a term in Q'.

The following theorem is the main result in this section and establishes that the EqNPE Algorithm 1 reaches the B-closedness condition upon termination, independently from the narrowing strategy, unfolding rule, and abstraction operator. This is a key property of partial evaluation frameworks that is necessary to achieve completeness of the specialization.

Lemma 6.3. *Let $\mathscr{R} = (\Sigma, B, \vec{E})$ be a decomposition of an equational theory $(\Sigma, \mathscr{E})$, $\mathscr{S}$ a narrowing strategy, and Q a set of terms. If* EqNPE$(\mathscr{R}, Q, \mathscr{S})$ *terminates computing the set Q' of terms, then: (1) Q is B-closed w.r.t. Q', and (2) also the rules in the resulting partially evaluated theory $\mathscr{R}'$ are B-closed w.r.t. Q'.*

In order to ensure the termination of the algorithm, the partial narrowing trees must be finite and the iterative construction of the partial trees must eventually terminate while still guaranteeing that the desired amount of specialization is retained and that the equational closedness condition is reached. In the following section, we recall a simple but useful solution to the termination problem by introducing appropriate unfolding and abstraction functions that fit the narrowing strategies described in Chapter 2 for specializing equational theories.

6.2.4 Termination of the PE process

Partial evaluation involves two classical termination problems: the so-called *local* termination problem (the termination of unfolding, or how to control and keep the expansion of the narrowing trees finite, which is managed by an unfolding rule), and *global* termination (which concerns termination of recursive unfolding, or how to stop recursively constructing more and more narrowing trees).

The problem of obtaining (sensibly expanded) finite narrowing trees essentially boils down to define sensible unfolding rules that somehow ensure that infinite unfolding is not performed. In the following section, we introduce an unfolding rule that attempts to maximize unfolding while retaining termination. In this thesis, we use the high-performance implementation of the order-sorted symbolic homeomorphic embedding relation $\check{\trianglelefteq}_B$ of [Alpuente et al. 2019b] that is summarized in Chapter 4, where a comparison of increasingly efficient implementations of $\check{\trianglelefteq}_B$ can be found. State of the art local control rules based on homeomorphic embedding do not check for embedding against all previously selected expressions but rather only against those in its sequence of *covering ancestors* [Bruynooghe et al. 1991]. This increases both the efficiency of the checking and whistling later. The following unfolding function makes use of the embedding relation in a constructive way to produce finite narrowing trees and then extract the leaves from the trees.

We need the following auxiliary notion. We say that a narrowing derivation $\mathscr{D}$ is *admissible w.r.t.* $\check{\trianglelefteq}_B$ if and only if it does not contain a pair of comparable narrowing redexes (i.e., rooted by the same operation symbol) s and t, where s precedes t in $\mathscr{D}$, such that $s \check{\trianglelefteq}_B t$.

Definition 6.4 (Unfolding function)**.** Given the rewrite theory $\mathscr{R} = (\Sigma, B, \vec{E})$ and a term t_0 to be specialized in $\mathscr{R}$, we define UNFOLD(t_0,$\mathscr{R}$,$\mathscr{S}$), for $\mathscr{S} = VN^{\circlearrowleft}_{\mathscr{R}}$, as the set of terms

$$\begin{aligned} Unfold^{\check{\trianglelefteq}_B}(t_0,\mathscr{R}) = \{t_n \mid\ & t_0 \leadsto^n t_n \in VN^{\circlearrowleft}_{\mathscr{R}}(t_0), \\ & t_0 \leadsto^{n-1} t_{n-1} \text{ is admissible w.r.t. } \check{\trianglelefteq}_B \text{ and} \\ & \text{either } \nexists w : t_0 \leadsto^n t_n \leadsto w \in VN^{\circlearrowleft}_{\mathscr{R}}(t_0) \\ & \text{or } t_0 \leadsto^n t_n \text{ is not admissible w.r.t. } \check{\trianglelefteq}_B.\} \end{aligned}$$

Given a set Q of terms, we also define $Unfold^{\check{\trianglelefteq}_B}(Q,\mathscr{R}) = \bigcup_{t\in Q} Unfold^{\check{\trianglelefteq}_B}(t,\mathscr{R})$.

Hence derivations are stopped when there is no further folding variant narrowing steps or the embedding whistle blows.

Example 6.6 (Example 6.5 continued)**.** *Consider again the (partial) folding variant narrowing tree of Figure 6.6. The narrowing redex*

$$t = \texttt{flip(fix(2, e, flip(BG')) ; \{R2 I R1\})}$$

in the right branch of the tree embeds modulo ACU the tree root

$$u = \texttt{flip(fix(2, e, flip(BG)))}$$

Since the whistle $u \check{\trianglelefteq}_B t$ blows, the unfolding of this branch is stopped.

The following result establishes the termination of the unfolding process.

Theorem 6.5 (Local Termination)**.** *Let $\mathscr{R} = (\Sigma, B, \vec{E})$ be a decomposition of an equational theory $(\Sigma, E \uplus B)$ and Q be a finite set of terms. The computation of $Unfold^{\check{\trianglelefteq}_B}(Q,\mathscr{R})$ terminates.*

Nontermination of the EqNPE algorithm can be caused not only by the creation of an infinite narrowing tree but also by never reaching the equational closedness condition. Unlike local control, which is parametric w.r.t. the decision whether to stop or to proceed with the expansion, since it is safe to terminate the evaluation at any point, the global control does not allow this flexibility because we cannot stop the iterative extension of the set Q of partially evaluated expressions until all function calls in this set are B-closed w.r.t. Q itself.

6.2.5 Global Termination of Equational NPE

For global termination, partial evaluation relies on an abstraction operator to ensure that the iterative construction of a sequence of partial narrowing trees terminates while still guaranteeing that the desired amount of specialization is retained and that the equational closedness condition is reached. In order to avoid constructing infinite sets, instead of just taking the union of the set $\mathscr{L}$ of (possibly non-closed modulo B) terms in the leaves of the tree and the set Q of specialized calls, the sets Q and $\mathscr{L}$ are *generalized*. Hence, the abstraction operator returns a safe approximation A of $Q \cup \mathscr{L}$ so that each expression in the set $Q \cup \mathscr{L}$ is closed w.r.t. A. Let us show how we can define a suitable abstraction operator by using the notion of *equational least general generalization* modulo B (lgg_B) [Alpuente et al. 2014b] so that we do not lose too

much precision despite the abstraction. For more sophisticated global control, homeormorphic embedding can be combined with other techniques such as global trees, characteristic trees, or trace terms (see, e.g., [Leuschel and Bruynooghe 2002] and its references).

For order–sorted theories, neither more general unifiers (mugs) nor least general generalizers (lggs) are generally unique, but there are always finite sets of them [Alpuente et al. 2014b]. In [Alpuente et al. 2014b], the notion of least general generalization is extended to work modulo (order-sorted) equational axioms B, where function symbols can obey any combination of associativity, commutativity, and identity axioms (including the empty set of such axioms). Unlike the untyped case, there is in general no unique lgg in the framework of [Alpuente et al. 2014b], due to both the order-sortedness and to the equational axioms. Instead, there is always a finite, minimal and complete set of lggs so that any other generalizer has at least one of them as a B-instance.

Given the current set Q of already specialized calls, in order to augment Q with a new set T of terms, the abstraction operator $\text{ABSTRACT}(Q,T,B)$ of Algorithm 1 is particularized to an abstraction function that relies on the notion of *best matching set* (BMS), an order-sorted equational extension of [Albert et al. 1998] that is aimed at avoiding loss of specialization due to generalization. The notion of BMS is used in the abstraction process when selecting the most appropriate terms of Q to be selected for generalizing T, in the sense of providing least general generalizations.

Example 6.7. [Albert et al. 1998] *Let $Q = \{f(g(x)), f(g(a), f(z)\}$ and $t = f(g(b))$. To compute the best matching set for t in Q, we first consider the set*

$$\begin{aligned} W &= \{lgg(\{f(g(x)), f(g(b))\}), lgg(\{f(g(a)), f(g(b))\}), lgg(\{f(z), f(g(b))\})\} \\ &= \{f(g(x)), f(g(y)), f(z)\} \end{aligned}$$

Now, the minimally general elements of W are $f(g(x))$ and $f(g(y))$, and thus we have $BMS_B(Q,t) = \{f(g(x)), f(g(a))\}$.

In this thesis, we determine the best matching set for t in a set U of terms w.r.t. B, $BMS_B(U,t)$, as follows: for each u_i in U, we compute the set $W_i = lgg_B(\{u_i,t\})$ and select the subset M of minimal upper bounds of the union $\bigcup_i W_i$. Then, the term u_k belongs to $BMS_B(U,t)$ if at least one element in the corresponding W_k belongs to M.

Definition 6.6 (Best Matching Set modulo B)**.** Let $U = \{u_1, \ldots, u_n\}$ be a set of terms and t be a term. Given the decomposition $(\Sigma, B, \vec{E})$ of $(\Sigma, E \uplus B)$, consider the sets of terms $W_i = \{w \mid \langle w, \{\theta_1, \theta_2\}\rangle \in lgg_B(\{u_i,t\})\}$, for $i = 1,..,n$, and $W = \bigcup_{i=1}^{n} W_i$. The *best matching set* $BMS_B(U,t)$ for t in U modulo B is the set of those terms $u_k \in U$ such that the corresponding W_k contains a *minimally general* element w of W under $\leq_B$, i.e., there is no different element w' in W (modulo the relation $\simeq_B$ induced by $\leq_B$) such that $w <_B w'$.

The following example illustrates the above definition.

Example 6.8. *Let $t \equiv g(1) \oplus 1 \oplus g(Y)$, $U \equiv \{1 \oplus g(X), X \oplus g(1), X \oplus Y\}$, and consider B to consist of the associativity and commutativity axioms for $\oplus$. To compute the best matching set for t in U, we first compute the sets of lgg_B's of t with each of the terms in U:*

$$W_1 = lgg_{AC}(\{g(1)\oplus 1\oplus g(Y), 1\oplus g(X)\}) = \{\langle\{Z\oplus 1\},\{Z/g(1)\oplus g(Y)\},\{Z/g(X)\}\rangle,$$
$$\langle\{Z\oplus g(W)\},\{Z/1\oplus g(1), W/Y\},\{Z/1, W/X\}\rangle\}$$
$$W_2 = lgg_{AC}(\{g(1)\oplus 1\oplus g(Y), X\oplus g(1)\}) = \{\langle\{Z\oplus g(1)\},\{Z/g(1)\oplus g(Y)\},\{Z/X\}\rangle\}$$
$$W_3 = lgg_{AC}(\{g(1)\oplus 1\oplus g(Y), X\oplus Y)\}) = \{\langle\{Z\oplus W\},\{Z/1, W/g(1)\oplus g(Y)\},\{Z/X, W/Y\}\rangle\}$$

Now, the set M of minimal upper bounds of the set $W_1\cup W_2\cup W_3$ is $M = \{Z\oplus 1, Z\oplus g(1)\}$ and thus we have: $BMS_{AC}(S,t) = \{1\oplus g(X), X\oplus g(1)\}$.

Now we are able to instantiate the function ABSTRACT(Q,T,B) of Algorithm 1 with the following equational abstraction function $abs^{\check{\trianglelefteq}_B}(Q,T)$ that relies on $\check{\trianglelefteq}_B$, the notion of best matching set, and equational least general generalization.

Definition 6.7 (Equational Least General Abstraction Function)**.** Let Q,T be two sets of terms. We define $abs^{\check{\trianglelefteq}_B}(Q,T)$ as follows:

$$\begin{cases} abs^{\check{\trianglelefteq}_B}(\ldots abs^{\check{\trianglelefteq}_B}(Q,\{t_1\}),\ldots,\{t_n\}) & \text{if } T\equiv\{t_1,\ldots,t_n\}, n>0 \\ Q & \text{if } T\equiv\emptyset \text{ or } T\equiv\{X\}, \textit{with } X\in\mathscr{X} \\ abs^{\check{\trianglelefteq}_B}(Q,\{t_1,\ldots,t_n\}) & \text{if } T\equiv\{t\}, \textit{with } t\equiv c(t_1,\ldots,t_n),\ c\in\mathscr{C} \\ generalize_B(Q,Q',t) & \text{if } T\equiv\{t\}, \textit{with } t\equiv f(t_1,\ldots,t_n),\ f\in\mathscr{D} \end{cases}$$

where $Q' = \{t'\in Q \mid root(t) = root(t') \text{ and } t'\check{\trianglelefteq}_B t\}$, and the function *generalize* is:

$$\begin{aligned} generalize_B(Q,\emptyset,t) &= Q\cup\{t\} \\ generalize_B(Q,Q',t) &= Q \text{ if } t \text{ is } B\text{-closed w.r.t. } Q \\ generalize_B(Q,Q',t) &= abs^{\check{\trianglelefteq}_B}(Q\setminus BMS_B(Q',t), Q''\!\downarrow_{\vec{E},B}) \quad \text{(otherwise)} \end{aligned}$$

where $Q''=\{l \mid q\in BMS_B(Q',t), \langle w,\{\theta_1,\theta_2\}\rangle\in lgg_B(\{q,t\}), x\in\mathscr{D}om(\theta_1\cup\theta_2),\ l\in\{w, x\theta_1, x\theta_2\}\}$.

Roughly speaking, the equational least general abstraction function proceeds as follows. We distinguish the cases when the considered term t either: i) is a variable, or ii) is not a variable. In the first case, the term is simply ignored. In the second case, if t does not B-embed any term in Q, it is just added. However, if t B-embeds some *comparable* term in Q, we distinguish two cases. If t is already Q-closed, then it is simply discarded. Otherwise, the given term is generalized by computing the lgg$_B$ of t w.r.t. each of its best matching terms, say q, that is $lgg_B(t,q) = \langle w,\{\theta_1,\theta_2\}\rangle$, and the abstraction function is recursively applied to add the B-normalized version of w and of the terms in the matching substitutions θ_1 and θ_2.

The following results establish the correctness and termination of the equational least general abstraction function.

Proposition 6.8. *The function $abs^{\check{\trianglelefteq}_B}$ of Definition 6.7 is an abstraction operator in the sense of Definition 6.2.*

Theorem 6.9. *The equational least general abstraction function $abs^{\check{\trianglelefteq}_B}$ terminates.*

Finally, the main result of this section follows from Theorem 6.5 and Proposition 6.8.

Theorem 6.10 (Global Termination). *Algorithm 1 terminates for the unfolding function $Unfold^{\check{\trianglelefteq}_B}$ and the equational least general abstraction function $abs^{\check{\trianglelefteq}_B}$.*

Let us illustrate the use of $abs^{\check{\trianglelefteq}_B}$ in the following specialization problem.

Example 6.9. *Let us consider again Example 6.8 and assume $Q \equiv \{1 \oplus g(X)\}$ and $T \equiv \{g(1) \oplus 1 \oplus g(Y)\}$. The call $abs^{\check{\trianglelefteq}_B}(Q,T)$ invokes*

$$generalize_B(Q,Q',g(1)\oplus 1\oplus g(Y))$$

with $Q' = \{1 \oplus g(X)\}$, which in turn calls $abs^{\check{\trianglelefteq}_B}(Q \setminus BMS_B(Q',t),Q'')$, where

$$\begin{array}{ll} BMS_B(Q',t) = & \{1\oplus g(X)\} \\ Q'' = & \{Z\oplus 1, Z\oplus g(W), g(X), g(1)\oplus g(Y), 1\oplus g(Y), 1\oplus g(X)\} \end{array}$$

This in turn calls to $abs^{\check{\trianglelefteq}_B}(\emptyset,Q'')$, which amounts to the sequence of imbricated calls

$$abs^{\check{\trianglelefteq}_B}(abs^{\check{\trianglelefteq}_B}(\emptyset, Z\oplus 1), \{Z\oplus g(W), g(X), g(1)\oplus g(Y), 1\oplus g(y), 1+g(X)\})$$

and since terms $Z\oplus 1, Z\oplus g(W)$ and $g(X)$ do not embed $1\oplus g(X)$, then the three terms are added yielding the new call

$$abs^{\check{\trianglelefteq}_B}(\{Z\oplus 1, Z\oplus g(W), g(X)\}, \{g(1)\oplus g(Y), 1\oplus g(y), 1+g(X)\})$$

that returns the set $\{Z\oplus 1, Z\oplus g(W), g(X)\}$ since all three terms $g(1)\oplus g(Y), 1\oplus g(y)$ and $1+g(X)$ are AC-closed. That is, $abs^{\check{\trianglelefteq}_B}(Q,T) = \{Z\oplus 1, Z\oplus g(W), g(X)\}$.

Example 6.10 (Example 6.6 continued). *Consider again the (partial) folding variant narrowing tree of Figure 6.6 with the leaf $t =$* `flip(fix(2, e, flip(BG') ; {R2 I R1}))` *in the right branch of the tree and the tree root $u =$* `flip(fix(2, e, flip(BG)))`*. We apply the equational least general abstraction function with $Q = \{u\}$ and $T = \{t\}$.*

Since t is operation-rooted, we call $generalize_B(Q,Q',t)$ with $Q' = Q$, which in turn calls to $abs^{\check{\trianglelefteq}_B}(Q \setminus BMS_{ACU}(Q',t),Q'')$, with $BMS_{ACU}(Q',t) = Q$ and $Q'' = \{w,v\}$, where $w =$ `flip(fix(2,e,flip(BG);BG'))` *is the only ACU least general generalization of u and t and $v =$* `{R2' I' R1'}`*. Then the call returns the set $\{w\}$. However, this means that the previous folding narrowing tree of Figure 6.6 is now discarded, since the previous set of input terms $Q = \{u\}$ is now replaced by $Q' = \{w\}$.*

We start from scratch and the tree results for the new call w is shown in Figure 6.7. The right leaf embeds the root of the tree and is B-closed w.r.t. it. The left leaf `mt` *is a constructor term. For the middle leaf $t'' =$* `{R2 I R1} ; flip(Bg''')` *the whistle* `flip(Bg'')` $\check{\trianglelefteq}_{ACU}$ t'' *blows and we stop the derivation. However, it is not B-closed w.r.t. w and we have to add it to the set Q', obtaining the new set of input terms $Q'' = \{w,$*`flip(Bg''')`*$\}$. The specialization of the call* `flip(Bg''')` *amounts to constructing the folding variant narrowing tree of Figure 6.8, which is trivially ACU-closed w.r.t. its root.*

Example 6.11 (Example 6.10 continued). *Since the two trees in Figures 6.7 and 6.8 do represent all possible computations for (any ACU-instance of) $u =$* `flip(fix(2, e, flip(BG)))`*, the partial evaluation process ends. Actually, u is an instance of the root of the tree in Figure 6.7 with* `{Bg' ↦ mt}` *because of the identity axiom. The computed specialization is the set Q'''.*

Now we can extract the set of resultants $t\sigma \Rightarrow r$ associated to the root-to-leaf derivations in the two trees, which yields:

```
eq flip(fix(2, e, flip(mt))) = mt .
eq flip(fix(2, e, flip({R1 I R2} ; BG’’’))) =
    flip(fix(2, e, flip(BG’’) ; {R2 I R1})) .
eq flip(fix(2, e, flip(mt) ; mt)) = mt .
eq flip(fix(2, e, flip(mt) ; BG ; {R1 I R2})) =
   {R2 I R1} ; flip(BG) .
eq flip(fix(2, e, flip({R1 I R2} ; BG) ; BG’)) =
   flip(fix(2, e, flip(BG) ; {R2 I R1} ; BG’)) .
eq flip(mt) = mt .
eq flip(BG ; {R1 I R2}) = {R2 I R1} ; flip(BG) .
```

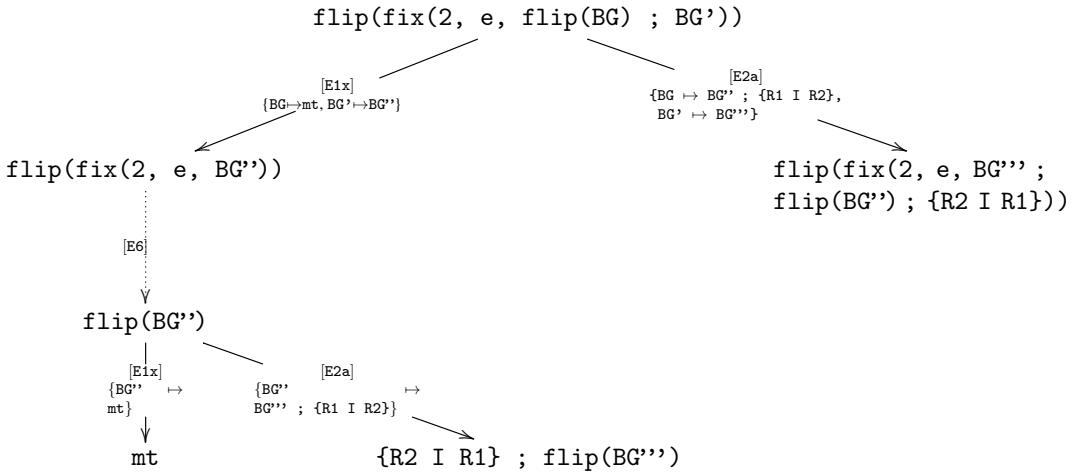


FIGURE 6.7: Folding variant narrowing tree for the goal `flip(fix(2, e, flip(BG) ; BG’))`.

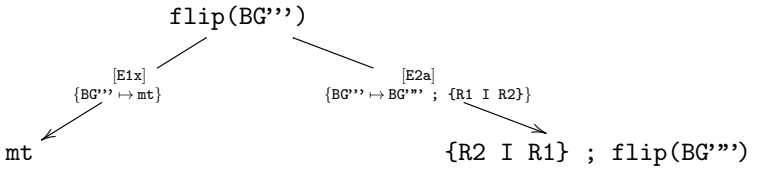


FIGURE 6.8: Folding variant narrowing tree for the goal `flip(BG’’’)`.

The reader may have realized that the specialization call `flip(fix(2,e,flip(BG)))` should really return the same term `BG`, since the variable `BG` is of sort `BinGraph` instead of `BinGraph?`, i.e., `flip(fix(2, e, flip(BG))) = BG`. The resultants above traverse the given graph and return the same graph. Though the code may seem inefficient, we have considered this example because it allows all the different stages of the PE process to be illustrated.

The following example shows how a better specialization program can be obtained.

Example 6.12. *Let us now overload the* `flip` *operator, having simultaneously two declarations for the* `flip` *symbol that are related in the subsort ordering* `Bingraph < Bingraph?`*:*

```
op flip : BinGraph -> BinGraph .
op flip : BinGraph? -> BinGraph? .
```

and four equations: E1, E2, E2a, *and* E2b. *By specializing the call* $t =$ `flip(fix(2,e,flip(BG)))`, *the subtype definition of* `flip` *allows Maude to simplify the term* t *using equation* E6, *which eliminates the occurrence of the* `fix` *symbol. All the leaves in the narrowing tree for* t, *shown in Figure 6.9, are B-closed w.r.t. the set of calls* {`flip(fix(2,e,flip(BG)))`,`flip(flip(BG'))`}. *This leads to the following, optimal specialized equations:*

```
eq flip(fix(2,e,flip(mt))) = mt .
eq flip(fix(2,e,flip({R1 I R2} ; BG))) = {R1 I R2} ; flip(flip(BG)) .
eq flip(flip(mt)) = mt .
eq flip(flip({R1 I R2} ; BG)) = {R1 I R2} ; flip(flip(BG)) .
```

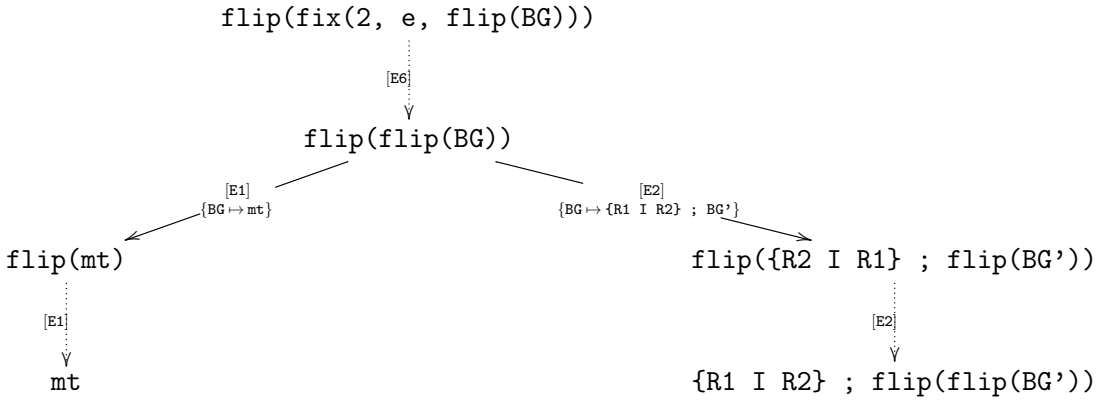


FIGURE 6.9: Folding variant narrowing tree for the goal `flip(fix(2, e, flip(BG)))`.

The use of (folding) variant narrowing during partial evaluation provides good overall behavior regarding both the elimination of intermediate data structures and the propagation of information. Moreover, the following result establishes that the executability requirements imposed on the original theory are preserved by the transformation; e.g., no infinite o diverging computations are encoded in the residual program.

Theorem 6.11. *The PE of a decomposition* $(\Sigma, B, \vec{E})$ *is a decomposition.*

Furthermore, in the following section, we extend the classical post-processing transformation [Alpuente et al. 1998a] to the order-sorted case modulo axioms to deliver a final partially evaluated program without any redundant or undesirable derivation that could not be proven in the original program.

6.2.6 Post-processing renaming modulo axioms

The basic PE algorithm of Section 6.2 incorporates only the basic scheme of a complete partial evaluator. The resulting partial evaluations might be further optimized by eliminating redundant function symbols and unnecessary repetition of variables. Essentially, we introduce a new function symbol for each specialized term and then replace each call in the specialized program by a call to the corresponding renamed function.

Definition 6.12 (Independent renaming [Alpuente et al. 1998a])**.** An independent renaming ρ for a set of terms T is a mapping from terms to terms defined as follows: for $t \in T$ with $root(t) =$

f being a function symbol, $\rho(t) = f_t(\overline{x_n})$, where $\overline{x_n}$ are the distinct variables in t in the order of their first occurrence and f_t is a new function symbol, which does not occur in $\mathscr{R}$ or T and is different from f and the root symbol of any other $\rho(t')$, with $t' \in T$ and $t' \neq t$. By abuse, we let $\rho(T)$ denote the set $T' = \{\rho(t) \mid t \in T\}$.

The renaming post-processing can be formally defined as follows.

Definition 6.13 (Post-partial evaluation)**.** Let T a finite set of terms and $\mathscr{R}'$ the (rewrite theory computed as a) partial evaluation of the rewrite theory $\mathscr{R}$ w.r.t. T s.t. $\mathscr{R}'$ is T-closed modulo B. Let ρ be an independent renaming for T. We define the post-partial evaluation $\mathscr{R}''$ of $\mathscr{R}$ w.r.t. T (under ρ) as follows:

$$\mathscr{R}'' = \bigcup_{t \in T} \{\rho(t)\theta \to ren_\rho(r) \mid t\theta \to r \in \mathscr{R}'\}$$

where the nondeterministic renaming function ren_ρ is defined as follows:

$$ren_\rho(t) = \begin{cases} t & \text{if } t \in \mathscr{X} \\ c(\overline{ren_\rho(t_n)}) & \text{if } t = c(\overline{t_n}),\ c \in \mathscr{C} \\ \rho(u)\theta' & \text{if } \exists \theta, \exists u \in T \text{ such that } t =_B u\theta \text{ and} \\ & \quad \theta' = \{x \mapsto ren_\rho(x\theta) \mid x \in \mathscr{D}om(\theta)\} \\ t & \text{otherwise} \end{cases}$$

Note that, while the independent renaming suffices to rename the left-hand sides of resultants (since they are mere instances of the specialized calls), the right-hand sides are renamed by means of the auxiliary function ren_ρ, which recursively replaces each call in the given expression by a call to the corresponding renamed function (according to ρ).

Theorem 6.14. *The post-partial evaluation of a decomposition $(\Sigma, B, \vec{E})$ is a decomposition.*

Finally, we state and prove the strong correctness of our partial evaluation technique. The proof proceeds essentially as follows. First, we prove the soundness (resp. completeness) of the transformation, i.e., we prove that, for each answer computed by folding variant narrowing in the original (resp. specialized) program, there exists a more general answer (modulo B) in the specialized (resp. original) program. Then, by using the minimality of folding variant narrowing, we conclude the strong correctness of the method, i.e., the answers computed in the original and the partially evaluated programs coincide (modulo B).

Theorem 6.15 (Strong Correctness and Completeness of Post-partial Evaluation)**.** *Let $\mathscr{R} = (\Sigma, B, \vec{E})$ be a decomposition of an equational theory $(\Sigma, E \uplus B)$, u be a Σ-term, and Q be a finite set of Σ-terms. Let ρ be an independent renaming of Q, $u' = ren_\rho(u)$ and $Q' = ren_\rho(Q)$. Let $\mathscr{R}' = (\Sigma, B, \vec{E}')$ be a partial evaluation of $\mathscr{R}$ w.r.t. Q (under the renaming ρ). If $\vec{E}'$ and u' are closed modulo B w.r.t. Q', then $(u \leadsto^*_{\sigma, \vec{E}, B} v) \in VN^{\circlearrowleft}_{\mathscr{R}}(u)$ if and only if $(u' \leadsto^*_{\sigma', \vec{E}', B} v') \in VN^{\circlearrowleft}_{\mathscr{R}'}(u)$, where $v' =_B ren_\rho(v)$.*

Example 6.13 (Example 6.12 continued)**.** *Consider the following independent renaming for the specialized calls:*

$$\{\texttt{flip(flip(BG))} \mapsto \texttt{dflip(BG)}\,,\ \texttt{flip(fix(2,e,flip(BG)))} \mapsto \texttt{flix(BG)}\}$$

The post-processing renaming derives the renamed program

```
eq flix(mt) = mt .
eq flix({R1 I R2} ; BG) = {R1 I R2} ; dflip(BG) .
eq dflip(mt) = mt .
eq dflip({R1 I R2} ; BG') = {R1 I R2} ; dflip(BG') .
```

that is more efficient and readable than the specialized program before renaming.

6.2.7 Specializing the interpreter of an imperative programming language

As a final example, let us discuss several specializations of the interpreter of an imperative language whose implementation as a Maude equational theory is publicly available at the webpage of our tool, Victoria. The interpreter provides the standard semantics of a simple imperative language that transforms program configurations $P \mid M$, where M belongs to sort `Memory` and represents the program memory, and P is an imperative program that may contain assignment instructions, conditional statements, arithmetic expressions, and loops. For simplicity, the interpreter assumes that, in an initial configuration $P_0 \mid M_0$, the memory M_0 is initialized with default values for all the variables in the program P_0 and thus it contains pairs `[x,v]` for each program variable `x` in P_0.

Consider we want to specialize the interpreter w.r.t. the following input term configuration

```
x := 0 ; if (x = 0) then y := 0 fi ; skip | M
```

where the Maude variable M stands for an unspecified program memory and is the only *logic or symbolic variable* in the input term, whereas program variables `x` and `y` are handled as constants by the interpreter.

Our tool Victoria returns the following extremely specialized version of the interpreter for the given input term, which can be seen as a compiled version written in Maude

```
eq x := 0 ; if (x = 0) then y := 0 fi ; skip | M [x,N1] [y,N2]
= skip | M [x,0] [y,0] [variant] .
```

Given the independent renaming

$$\rho = \{\text{“}\texttt{x := 0 ; if (x = 0) then y := 0 fi ; skip | M}\text{”} \mapsto \texttt{f1(M)},\ \text{“}\texttt{skip | M [x,0] [y,0]}\text{”} \mapsto \texttt{f2(M)}\}$$

the final, renamed version of the program is

```
eq f1(M [x,N1] [y,N2]) = f2(M) [variant]  .
```

Let us now consider the case when the interpreter is specialized w.r.t. a more interesting configuration $P \mid M$, where there is a second logic variable N (in addition to M) that appears in P and belongs to sort `Nat` of natural numbers.

```
x := 2 ; i := N ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ; skip | M
```

In this case, Victoria returns the following specialized interpreter

```
eq x := 2 ; i := 0 ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | M [c,N1] [i,N2] [x,N3]
  = skip | M [c,3] [i,2] [x,2] [variant] .
eq x := 2 ; i := 1 ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | M [c,N1] [i,N2] [x,N3]
  = skip | M [c,2] [i,2] [x,2] [variant] .
eq x := 2 ; i := 2 + N ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | M [c,N1] [i,N2] [x,N3]
  = skip | M [c,0] [i,2 + N] [x,2] [variant] .
```

Given the independent renaming

$\rho = \{$“x := 2 ; i := N; c := 0 ;
while (i < x) do i := i + 1 ; c := c + i od ; skip | M”$\mapsto$ f1(N,M),
“skip | M [c,0] [i,N] [x,2]”$\mapsto$ f2(N,M)$\}$

the final, renamed version of the program is

```
eq f1(0, M [c,N1] [i,N2] [x,N3]) = f2(0,M) [variant] .
eq f1(1, M [c,N1] [i,N2] [x,N3]) = f2(1,M) [variant] .
eq f1(2 + N, M [c,N1] [i,N2] [x,N3]) = f2(2 + N,M) [variant] .
```

Furthermore, if we make the memory of the previous input configuration more concrete (without any logic variable M)

```
x := 2 ; i := N ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | [c,0] [i,0] [x,0]
```

then our tool returns a simpler version of the same specialized program

```
eq f1(0) = f2(0) [variant] .
eq f1(1) = f2(1) [variant] .
eq f1(2 + N) = f2(2 + N) [variant] .
```

Note that this specialized program has no axiom, since the memory was defined as a multiset using an ACU symbol and it has been completely eliminated.

However, consider we specialize the interpreter w.r.t. the following symbolic configuration

```
x := N ; i := 0 ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | [c,0] [i,0] [x,0]
```

where the logical variable `N` occurs in the assignment for the program variable `x` that controls the number of loop iterations. Then, our tool returns the following specialized version of the interpreter

```
eq x := 0 ; i := 0 ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | [c,0] [i,0] [x,0]
  = skip | [c,0] [i,0] [x,0]  [variant] .
eq x := 1 + N ; i := 0 ; c := 0 ;
  while (i < x) do i := i + 1 ; c := c + i od;
  skip | [c,0] [i,0] [x,0]
  = if (i < x) then i := i + 1 ; c := c + i fi;
    while (i < x) do i := i + 1 ; c := c + i od ;
    skip | [c,1] [i,1] [x,1 + N] [variant] .
eq if (i < x) then i := i + 1 ; c := c + i fi ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | [c,N1] [i,N2 + 1 + N3] [x,N2] .
  = skip | [c,N1] [i,N2 + 1 + N3] [x,N2] [variant] .
eq if (i < x) then i := i + 1 ; c := c + i fi ;
  while (i < x) do i := i + 1 ; c := c + i od ;
  skip | [c,N1] [i,N2] [x,N2 + 1 + N3]
  = if (i < x) then i := i + 1 ; c := c + i fi ;
    while (i < x) do i := i + 1 ; c := c + i od ;
    skip | [c,N1 + N2] [i,N2 + 1] [x,N2 + 1 + N3] [variant] .
```

Given the independent renaming

$$\rho = \{\text{“x := N; i := 0; c := 0 ; while (i < x) do i := i + 1 ; c := c + i od ; skip | [c,0] [i,0] [x,0]”} \mapsto \texttt{f1(N)},$$
$$\text{“skip | M [c,N1] [i,N2] [x,N3]”} \mapsto \texttt{f2(N1,N2,N3)},$$
$$\text{“if (i < x) then i := i + 1; c := c + i fi ; while (i < x) do i := i + 1 ; c := c + i od ; skip | [c,N1] [i,N2] [x,N3]”} \mapsto \texttt{f3(N1,N2,N3)}\}$$

the final, renamed version of the program is

```
eq f1(0) = f2(0,0,0) [variant] .
eq f1(1 + N) = f3(1,1,1 + N) [variant] .
eq f3(N1,N2 + 1 + N3,N2) = f2(N1,N2 + 1 + N3,N2) [variant] .
eq f3(N1,N2,N2 + 1 + N3) = f3(N1 + N2,N2 + 1,N2 + 1 + N3) [variant] .
```

Note that this specialization does not offer much improvement over the original interpreter, as expected because the while loop has been unrolled only once.

Interestingly, the specialization time is negligible in all these examples thanks to the (order-sorted) equational least general generalization of [Alpuente et al. 2019b] and the homeomorphic embedding modulo equational axioms of [Alpuente et al. 2019a]. This in contrast to our previous prototype tool in [Alpuente et al. 2017a], which exceeded a generous timeout of several hours (similar to the specialization times for a comparable language interpreter in [Leuschel et al. 2007]). Furthermore, the size of the specialized program (after renaming) is less than 10% of the size of the original interpreter.

6.3 Experiments

We have implemented and experimentally evaluated the transformation framework presented in this section in the automatic Maude partial evaluator Victoria [Victoria Website].

Table 6.1 contains the experiments that we have performed using an Intel Core2 Quad CPU Q9300(2.5GHz) with 6 Gigabytes of RAM running Maude v2.7 and considering the average of ten executions for each test. These experiments together with the source code of all examples are publicly available. We have considered the following four programs: (i) Parser (Example 1.5), (ii) Double-flip (Example 6.1), (iii) Flip-fix (Example 6.2), and (iv) an implementation of the interpreter for an imperative language (the example discussed in Section 6.2.7). We have also considered the classical KMP string pattern matcher [Alpuente et al. 1998a]. For all five Maude programs, we consider input data of three different sizes: one hundred thousand elements, one million elements, and five million elements. Here elements refer to graph nodes for Double-flip and Flip-fix, and list elements for Parser and KMP. We have benchmarked three versions of each program on these data: original program, partially evaluated program (before post-processing renaming), and final specialization (with post-processing renaming). We do not explicitly show the specialization times since they are negligible for all problems ($<$ 100 ms.), which means it is up to 6 orders of magnitude faster than the preliminary prototype tool in [Alpuente et al. 2017a].

		Original	**PE before renaming**		**PE after renaming**	
Benchmark	**Data**	**Time (ms)**	**Time (ms)**	**Speedup**	**Time (ms)**	**Speedup**
Parser	100k	164	39	76,22	33	79,88
	1M	10.561	411	96,11	348	96,70
	5M	275.334	2.058	99,25	1.685	99,39
Double-flip	100k	188	143	23,94	76	59,57
	1M	1.636	1.427	12,78	759	53,61
	5M	8.425	7.503	10,94	4.100	51,34
Flip-fix	100k	203	177	12,81	143	29,56
	1M	1.955	1.778	9,05	1.427	27,01
	5M	10.185	9.219	9,48	7.458	26,77
KMP	100k	401	57	87,79	36	91,02
	1M	3.872	531	86,29	331	91,45
	5M	19.932	2.530	87,31	1.661	91,67
Interpreter	1k	5	3	40,00	2	60,00
	10k	53	22	22,64	12	41,51
	100k	520	248	21,54	112	47,69

TABLE 6.1: Experimental results

The relative speedups that we achieved thanks to specialization are given in the Speedup column(s) and computed as the percentage $100 \times (\mathsf{OriginalTime} - \mathsf{PETime})/\mathsf{OriginalTime}$. For all of the examples, the partially evaluated programs achieve a significant improvement in execution time when compared to the original program, both with and without renaming, but even more noticeable after renaming. The average improvement for these benchmarks is 66.5%. Of course, the bigger input goals give larger speedups. Often, the price paid is the size of the residual program, which may grow linearly with the size of the specialized call. For the KMP test, the average improvement is 91.67%. That is, the achieved speedup is 12 ($OriginalTime/PETime$), which is comparable to the average speedup of 14 that is achieved by both the CPD-based partial evaluator ECCE [Jørgensen et al. 1996] and the PE tool of [Albert et al. 2002] (actually, the generated residual programs are identical to [Albert et al. 2002] on this benchmark). This indicates that our new partial evaluation scheme is a conservative extension of previous approaches on comparable examples.

Moreover, matching modulo axioms such as associativity, commutativity, and identity are fairly expensive operations that are massively used in Maude, which can sometimes be drastically reduced after specialization. For instance, when we specialized the Parser of Example 1.5 using our tool Victoria, as illustrated in Figure 1.7, it moves from a program with ACU and U-right operators to a program without axioms, as illustrated in Figure 1.8. This transformation power cannot be achieved by traditional NPE nor competing on-line partial evaluation techniques, such as conjunctive partial deduction or positive supercompilation [Alpuente et al. 1998b].

Chapter 7

Conclusions

This thesis develops the *first partial evaluation framework* for order-sorted equational theories. The formalized scheme supports sorts, subsorts, subsort polymorphism, convergent rules (equations), and modular combinations of equational axioms. In the following we provide some concluding remarks and point out directions for future work.

First, we have laid the theoretical foundations for developing the first practical program specializer for Maude programs. Our specializer implements novel control criteria that ensure both local and global termination, which required classical partial evaluation notions to be correctly and efficiently generalized to the equational case; e.g., the order-sorted symbolic homeomorphic embedding modulo axioms and the order-sorted equational least general generalization that allow us to define equational closedness and abstraction. We have presented our implementation of the narrowing driven equational partial evaluator Victoria and have shown that it achieves a significant increase in speed on realistic examples. The development of a complete partial evaluator for the entire Maude language requires dealing with some features that are not considered in this work and experimenting with more refined heuristics that maximize the specialization power. Future implementation work will focus on: (i) extending the Equational NPE framework to deal with more complex rewrite theories that may include (conditional) rules, equations, and axioms; and (ii) developing refined heuristics that can lead to further optimizations under mild assumptions (e.g., based on identifying subtheories that enjoy the finite variant property and other commonly occurring properties). We believe that advancing current PE research ideas for order-sorted rewrite theories may open up new opportunities for optimization in rewriting logic program semantics development frameworks, e.g., [Rusu et al. 2016]. Also, besides serving as a powerful tool to boost program performance, it will also be a significant driver of new symbolic reasoning features in Maude and further improvements in Maude's narrowing infrastructure.

Second, the visualization of program executions has recently 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 finite variant property. Besides the applications outlined in this thesis, 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. 2009a] 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

2018], the partial evaluator Victoria [Alpuente et al. 2017a], 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 of 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. In order to give support to all these applications, as future work we plan to address several extensions of GLINTS, such as computing constructor variants [Meseguer 2018] and irredundant variants [Clavel et al. 2016], and supporting irreducibility constraints [Erbatur et al. 2012].

Third, we have presented ACUOS^2, a highly optimized implementation of the order-sorted ACU least general generalization algorithm formalized in [Alpuente et al. 2014b]. ACUOS^2 is a new, high-performance version of its predecessor ACUOS [Alpuente et al. 2014a]. ACUOS^2 runs up to five orders of magnitude faster than ACUOS and scales to real-life problem sizes in which ACUOS fails to give a response, such as the biomedical domains often addressed in inductive logic programming and other AI applications.

Finally, regarding the homeomorphic embedding it has been extensively used in Prolog, but it has never been investigated in the context of expressive rule-based languages like Maude, CafeOBJ, OBJ, ASF+SDF, and ELAN that support symbolic reasoning methods modulo equational axioms. We have introduced a new equational and order-sorted definition of homeomorphic embedding and efficiently implemented in HEMS, our equational homeomorphic embedding checker for Maude, which gives a remarkably good performance for theories with symbols satisfying any combination of associativity and/or commutativity axioms. We have also compared different definitions of embedding, identifying some key conclusions: (i) definitions of equational homeomorphic embedding based on (non-deterministic) search in Maude perform dramatically worse than their equational counterparts and are not feasible in practice; (ii) definitions of equational homeomorphic embedding based on generated theories perform dramatically worse than meta-level definitions; and (iii) the flattened meta-representation of terms is crucial for homeomorphic embedding definitions dealing with A and AC operators in order to pay off in practice. As future work, we plan to extend our results to the case when the equational theory B may contain the identity axiom, which is non-trivial since B is not class-finite.

Bibliography

ACUOS^2 Website, 2018. http://safe-tools.dsic.upv.es/acuos2.

H. Aït-Kaci and Y. Sasaki. An axiomatic approach to feature term generalization. In *Proc. of the 12th European Conference on Machine Learning (EMCL 2001)*, volume 2167 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of the 5th International Symposium on Static Analysis (SAS 1998)*, volume 1503 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 1998.

E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of the 6th International Conference on Logic Programming and Automated Reasoning (LPAR 1999)*, volume 1705 of *Lecture Notes in Computer Science*, pages 376–395. Springer, 1999.

E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002, 2002. URL http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-01.pdf.

M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In *Proc. of the 6th European Symposium on Programming (ESOP 1996)*, volume 1058 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 1996.

M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1997)*, pages 151–162. ACM, New York, 1997a.

M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In *Proc. of the International Conference on Algebraic and Logic Programming (ALP 1997)*, volume 1298 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Berlin, 1997b.

M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998a.

M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998b.

M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of inductively sequential functional logic programs. In *Proc. of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 1999), Paris, France, September 27-29, 1999*,

pages 273–283. ACM, 1999. ISBN 1-58113-111-9. doi: 10.1145/317636.317910. URL https://doi.org/10.1145/317636.317910.

M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance. In *Proc. of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014)*, volume 8761 of *Lecture Notes in Computer Science*, pages 573–581. Springer-Verlag, Berlin, 2014a.

M. Alpuente, S. Escobar, J. Espert, and J. Meseguer. A Modular Order-sorted Equational Generalization Algorithm. *Information and Computation*, 235:98–136, 2014b. doi: 10.1016/j.ic.2014.01.006. URL https://doi.org/10.1016/j.ic.2014.01.006.

M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Exploring Conditional Rewriting Logic Computations. *Journal of Symbolic Compututation*, 69:3–39, 2015. doi: 10.1016/j.jsc.2014.09.028. URL https://doi.org/10.1016/j.jsc.2014.09.028.

M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Assertion-based Analysis via Slicing with ABETS. 16(5–6):515–532, 2016.

M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. 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), Edinburgh, UK, September 6-8, 2016*, volume 10184 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017a. doi: 10.1007/978-3-319-63139-4_1. URL https://doi.org/10.1007/978-3-319-63139-4_1.

M. Alpuente, S. Escobar, J. Sapiña, and A. Cuenca-Ortega. Inspecting maude variants with GLINTS. *Theory and Practice of Logic Programming*, 17(5-6):689–707, 2017b. doi: 10.1017/S147106841700031X. URL https://doi.org/10.1017/S147106841700031X.

M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. Homeomorphic embedding modulo combinations of associativity and commutativity axioms. In *Proc. of the 28th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2018), furt/Main, Germany, September 4-6, 2018*, volume 11408 of *Lecture Notes in Computer Science*, pages 38–55. Springer, 2018. ISBN 978-3-030-13837-0. doi: 10.1007/978-3-030-13838-7_3. URL https://doi.org/10.1007/978-3-030-13838-7_3.

M. Alpuente, D. Ballis, A. Cuenca, S. Escobar, and J. Meseguer. ACUOS^2: A High-performance System for Modular ACU Generalization with Subtyping and Inheritance. In *Proc. of the 19th European Conference on Logics in Artificial Intelligence (JELIA 2019), Rende, Italy, May 7-11, 2019*, volume 11468 of *Lecture Notes in Artificial Intelligence*, pages 171–181. Springer, 2019a. doi: 10.1007/978-3-030-19570-0_11. URL https://doi.org/10.1007/978-3-030-19570-0_11.

M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. Order-sorted Homeomorphic Embedding modulo Combinations of Associativity and/or Commutativity Axioms. 2019b. Accepted for publication.

M. Alpuente, A. Cuenca-Ortega, S. Escobar, and J. Meseguer. A Partial Evaluation Framework for Order-sorted Equational Programs modulo Axioms. 2019c. Submitted for publication.

E. Armengol. Usages of Generalization in Case-Based Reasoning. In *Proc. of the 7th International Conference on Case-Based Reasoning (ICCBR 2007)*, volume 4626 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 2007. ISBN 978-3-540-74138-1. doi: 10.1007/978-3-540-74141-1_3.

E. Armengol and E. Plaza. Symbolic explanation of similarities in case-based reasoning. *Computers and Artificial Intelligence*, 25(2-3):153–171, 2006.

F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

F. Baader, W. Snyder, P. Narendran, M. Schmidt-Schauß, and K. U. Schulz. Unification theory. In *Handbook of Automated Reasoning*, volume 2, pages 445–532. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9. doi: 10.1016/b978-044450813-3/50010-2. URL https://doi.org/10.1016/b978-044450813-3/50010-2.

K. Bae, S. Escobar, and J. Meseguer. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *Proc. of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. A variant of higher-order anti-unification. In *Proc. of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

A. Baumgartner, T. Kutsia, J. Levy, and M. Villaret. Term-graph anti-unification. In *Proc. of the 3rd International Conference on Formal Structures for Computation and Deduction, (FSCD 2018), July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. ISBN 978-3-95977-077-4. doi: 10.4230/LIPIcs.FSCD.2018.9. URL https://doi.org/10.4230/LIPIcs.FSCD.2018.9.

C. Bouchard, K. A. Gero, C. Lynch, and P. Narendran. On Forward Closure and the Finite Variant Property. In *Proc. of the 9th International Symposium on Frontiers of Combining Systems (FroCos 2013)*, volume 8152 of *Lecture Notes in Computer Science*, pages 327–342. Springer-Verlag, Berlin, 2013.

M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs. *New Generation Comput.*, 11(1): 117–131, 1991.

H. Bürckert, A. Herold, and M. Schmidt-Schauß. On Equational Theories, Unification, and (Un)decidability. *Journal of Symbolic Computation*, 8(1–2):3–49, 1989.

R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013. ISSN 0001-0782.

W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

A. Cholewa, J. Meseguer, and S. Escobar. Variants of variants and the finite variant property. Technical report, CS Dept. University of Illinois at Urbana-Champaign, february 2014. URL http://hdl.handle.net/2142/47117.

N. H. Christensen and R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Trans. Program. Lang. Syst.*, 26(1):191–220, 2004.

M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002. doi: 10.1016/S0304-3975(01)00360-7. URL https://doi.org/10.1016/S0304-3975(01)00360-7.

M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer-Verlag, 2007.

M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and Narrowing in Maude 2.4. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil, June 29 - July 1, 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.7.1)*, July 2016. Available at: http://maude.cs.illinois.edu.

H. Comon-Lundh and S. Delaune. 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)*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005. ISBN 3-540-25596-6.

C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, and E. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Comput. Surv.*, 30(3es):19, 1998. doi: 10.1145/289121.289140. URL https://doi.org/10.1145/289121.289140.

W. R. Cook and R. Lämmel. Tutorial on Online Partial Evaluation. In *Proc. of the Working Conference on Domain-Specific Languages, DSL, (IFIP 2011), Bordeaux, France, 6-8th September 2011.*, volume 66 of *EPTCS*, pages 168–180, 2011.

J. Darlington, Y. Guo, and H. Pull. Constraints unify functional and logic programming. Technical report, Department of Computing, Imperial College, London, 1991.

P. Deransart, M. V. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2006. ISBN 978-3-540-40016-5.

N. Dershowitz. A Note on Simplification Orderings. *Information Processing Letters*, 9(5): 212–215, 1979. ISSN 00200190. doi: 10.1016/0020-0190(79)90071-1.

N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

F. Durán and J. Meseguer. A maude coherence checker tool for conditional order-sorted rewrite theories. In *Proc. of the 8th International Workshop on Rewriting Logic and Its Applications (WRLA 2010), Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2010. ISBN 978-3-642-16309-8. doi: 10.1007/978-3-642-16310-4_7. URL https://doi.org/10.1007/978-3-642-16310-4_7.

F. Durán and J. Meseguer. On the Church-Rosser and Coherence Properties of Conditional Order-sorted Rewrite Theories. 81(7–8):816–850, 2012.

F. Durán, S. S. Lucas, and J. Meseguer. Termination Modulo Combinations of Equational Theories. In *Proc. of the 7th International Symposium on Frontiers of Combining Systems (FroCoS 2009)*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer Berlin Heidelberg, 2009.

F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, and C. Talcott. 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)*, volume 9706 of *Lecture Notes in Computer Science*, pages 183–192. Springer-Verlag, Berlin, 2016.

S. Eker. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, 38(5):381, 1995. doi: 10.1093/comjnl/38.5.381. URL +http://dx.doi.org/10.1093/comjnl/38.5.381.

S. Eker. Single Elementary Associative-Commutative Matching. *J. Autom. Reasoning*, 28 (1):35–51, 2002. doi: 10.1023/A:1020122610698. URL https://doi.org/10.1023/A:1020122610698.

S. Eker. Associative-Commutative Rewriting on Large Terms. In *Proc. of the 14th International Conference on Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2003.

S. Erbatur, S. Escobar, D. Kapur, Z. Liu, C. Lynch, C. Meadows, J. Meseguer, P. Narendran, S. Santiago, and R. Sasse. Effective Symbolic Protocol Analysis via Equational Irreducibility Conditions. In *Proc. of the 17th European Symposium on Research in Computer Security (ESORICS 2012)*, volume 7459 of *Lecture Notes in Computer Science*, pages 73–90. Springer-Verlag, Berlin, 2012.

A. Ershov. Mixed Computation: Potential Applications and Problems for Study. *Theoretical Computer Science*, 18:41–67, 1982.

S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties. In *Foundations of Security Analysis and Design V (FOSAD 2007/2008/2009 Tutorial Lectures)*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, Berlin, 2009a.

S. Escobar, J. Meseguer, and R. Sasse. Variant Narrowing and Equational Unification. *Electronic Notes Theoretical Computer Science*, 238(3):103–119, 2009b.

S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 81(7-8):898–928, 2012.

M. Fay. First Order Unification in an Equational Theory. In *Proc of the 4th International Conf. on Automated Deduction (CADE 1979)*, pages 161–167, 1979.

J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.

H. Garavel, M. Tabikh, and I. Arrada. Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In *Proc. of the 12th International Workshop on Rewriting Logic and*

Its Applications (WRLA 2018), Held as a Satellite Event of ETAPS, Thessaloniki, Greece, June 14-15, 2018, volume 11152 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2018. ISBN 978-3-319-99839-8. doi: 10.1007/978-3-319-99840-4_1. URL https://doi.org/10.1007/978-3-319-99840-4_1.

D. Gentner. Structure-Mapping: A Theoretical Framework for Analogy*. *Cognitive Science*, 7 (2):155–170, 1983. ISSN 1551-6709. doi: 10.1207/s15516709cog0702_3.

GLINTS Experiments, 2017. http://safe-tools.dsic.upv.es/glints/pages/experiments.jsp.

GLINTS Quick Start Guide, 2017. http://safe-tools.dsic.upv.es/glints/download/quickstart.pdf.

GLINTS Website, 2017. http://safe-tools.dsic.upv.es/glints.

R. Glück and M. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. of the 7th International Symposium on Programming Language Implementation and Logic Programming (PLILP 1994)*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 1994.

J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105: 217–273, 1992.

M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

M. Hanus. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *Lecture Notes in Computer Science*, pages 123–168. Springer, 2013.

M. Hanus and B. Peemöller. A partial evaluator for curry. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335, pages 55–71. Universität Halle-Wittenberg, 2014.

HEMS Website, 2019. http://safe-tools.dsic.upv.es/hems.

R. Ji and R. Bubel. Pe-key: A partial evaluator for java programs. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *Integrated Formal Methods*, pages 283–295, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-30729-4.

N. Jones. Partial Evaluation, Self-Application and Types. In *Proc. of the 17th International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 639–659. Springer, 1990.

N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

J. Jørgensen, M. Leuschel, and B. Martens. Conjunctive Partial Deduction in Practice. In *Proc. of the 8th International Workshop on Logic Programming Synthesis and Transformation, (LOPSTR 1996)*, volume 1207 of *Lecture Notes in Computer Science*, pages 59–82. Springer, 1996.

J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental Construction of Unification Algorithms in Equational Theories. In *Proc. of 10th Colloquium on Automata, Languages and Programming (ICALP 1983)*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

H. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symposium on Principles of Programming Languages*, pages 255–267, 1982.

J. Kruskal. Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

L. Lafave and J. P. Gallagher. Constraint-based partial evaluation of rewriting-based functional logic programs. In *Proc. of the 7th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR 1997)*, volume 1463 of *Lecture Notes in Computer Science*, pages 168–188. Springer, 1998.

M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, 1997.

M. Leuschel. Improving Homeomorphic Embedding for Online Termination. In *Proc. of the 8th International Workshop on Logic Programming Synthesis and Transformation, (LOPSTR 1998)*, volume 1559 of *Lecture Notes in Computer Science*, pages 199–218. Springer, 1998a.

M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proc. of 5th International Symposium on Static Analysis, SAS'98*, volume 1503 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 1998b. ISBN 3-540-65014-8. doi: 10.1007/3-540-49727-7_14. URL http://dx.doi.org/10.1007/3-540-49727-7{_}14.

M. Leuschel. 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*, volume 2566 of *Lecture Notes in Computer Science*, pages 379–403. Springer-Verlag, Berlin, 2002.

M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *TPLP*, 2(4-5):461–515, 2002.

M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM TOPLAS*, 20(1):208–258, 1998.

M. Leuschel, S. Craig, and D. Elphick. Supervising Offline Partial Evaluation of Logic Programs Using Online Techniques. In *Proc. of the 16th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2006), Venice, Italy, July 12-14, 2006, Revised Selected Papers*, volume 4407 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2007.

J. Lloyd and J. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of Ithe 12th International Conference on Logic Programming (ICLP 1995)*, pages 597–611. MIT Press, 1995.

Mau-Dev Website, 2016. Available at: http://safe-tools.dsic.upv.es/maudev.

S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Proc. of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer-Verlag, Berlin, 2013.

P. Melliès. On a duality between Kruskal and Dershowitz theorem. In *The 25th International Colloquium on Automata, Languages, and Programming (ICALP 1998)*, volume 1443 of *Lecture Notes in Computer Science*, pages 518–529. Springer, 1998. ISBN 3-540-64781-3.

J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

J. Meseguer. Membership Algebra As a Logical Framework for Equational Specification. In *Proc. of the 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT 1997)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997. ISBN 3-540-64299-4.

J. Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81 (7-8):721–781, 2012. doi: 10.1016/j.jlap.2012.06.003. URL https://doi.org/10.1016/j.jlap.2012.06.003.

J. Meseguer. Order-Sorted Rewriting and Congruence Closure. In *Proc. of the 19th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2016)*, volume 9634 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2016.

J. Meseguer. Strict Coherence of Conditional Rewriting Modulo Axioms. *Theoretical Computer Science*, 672:1–35, 2017. doi: 10.1016/j.tcs.2016.12.026. URL http://dx.doi.org/10.1016/j.tcs.2016.12.026.

J. Meseguer. Variant-based satisfiability in initial algebras. *Science of Computer Programming*, 154:3–41, 2018. doi: 10.1016/j.scico.2017.09.001. URL https://doi.org/10.1016/j.scico.2017.09.001.

J. Meseguer and P. Thati. Symbolic Reachability Analysis using Narrowing and its Application to Verification of Cryptographic Protocols. *Higher-Order and Symbolic Computation*, 20 (1-2):123–160, 2007.

A. Middeldorp and B. Gramlich. Simple Termination is Difficult. *Applicable Algebra in Engineering, Communication and Computing*, 6(2):115–128, 1995. ISSN 09381279. doi: 10.1007/BF01225647.

S. Muggleton. Inductive Logic Programming: Issues, Results and the Challenge of Learning Language in Logic. *Artif. Intell.*, 114(1-2):283–296, 1999.

S. Ontañón and E. Plaza. Similarity measures over refinement graphs. *Machine Learning*, 87 (1):57–92, Apr. 2012. ISSN 0885-6125. doi: 10.1007/s10994-011-5274-3.

A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*, pages 355–385. Springer, 1996.

L. Pottier. Generalisation de termes en theorie equationelle: Cas associatif-commutatif. Technical Report INRIA 1056, Norwegian Computing Center, 1989.

V. Rusu, D. Lucanu, T. Serbanuta, A. Arusoaie, A. Stefanescu, and G. Rosu. Language definitions as rewrite theories. *Journal of Logic and Algebraic Methods in Programming*, 85(1): 98–120, 2016.

R. Singh and A. King. Partial Evaluation for Java Malware Detection. In *Proc. of the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2014), Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, volume 8981 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2014. ISBN 978-3-319-17821-9. doi: 10.1007/978-3-319-17822-6_8. URL https://doi.org/10.1007/978-3-319-17822-6_8.

J. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

M. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of the International Symposium on Logic Programming (ILPS 1995)*, pages 465–479. MIT Press, 1995.

M. Sørensen, R. Glück, and N. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

C. Talcott. Pathway logic. *Formal Methods for Computational Systems Biology*, 5016:21–53, 2008.

P. Thati and J. Meseguer. Complete symbolic reachability analysis using back-and-forth narrowing. *Theor. Comput. Sci.*, 366(1-2):163–179, 2006. doi: 10.1016/j.tcs.2006.07.008. URL https://doi.org/10.1016/j.tcs.2006.07.008.

V. Turchin. Program Transformation by Supercompilation. In *Programs as Data Objects, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 1986.

Victoria Website, 2019. http://safe-tools.dsic.upv.es/victoria.

P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM.

F. Yang, S. Escobar, C. Meadows, J. Meseguer, and P. Narendran. 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)*, pages 123–133. ACM Press, 2011.