Conditional Narrowing Modulo in Rewriting Logic and Maude

Luis Manuel Aguirre García

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de Fin de Máster en Programación y Tecnología Software

9 de septiembre de 2013

Dirigido por:

Narciso Martí Oliet
Miguel Palomino Tarjuelo
Isabel Pita Andreu

Autorización de difusión

Luis Manuel Aguirre García

9 de septiembre de 2013

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente trabajo de fin de máster: "Conditional Narrowing Modulo in Rewriting Logic and Maude", realizado durante el curso académico 2012-2013 bajo la dirección de Narciso Martí Oliet y Miguel Palomino Tarjuelo, y con la colaboración externa de dirección de Isabel Pita Andreu, en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

Este trabajo presenta un estudio sobre la resolución de problemas de alcanzabilidad en teorías de reescritura con una lógica ecuacional de pertenencia subyacente, usando la técnica de estrechamiento. Para ello se han desarrollado dos cálculos, uno que resuelve el problema de unificación módulo la lógica ecuacional y otro que resuelve el problema de alcanzabilidad basándose en el cálculo de unificación previo. Dichos cálculos hacen uso de algoritmos de pertenencia, de unificación módulo axiomas y de encaje, todos ellos disponibles en el lenguaje de reescritura Maude. En ambos cálculos se hace especial énfasis en el uso eficiente de la información sobre los tipos de los términos. Se ha demostrado la corrección y completitud de los cálculos. Posteriormente se han desarrollado sendos conjuntos de reglas de transformación para estos cálculos que permiten la implementación de los mismos. Finalmente, se han programado estas reglas en un prototipo, usando el lenguaje de reescritura Maude.

Palabras clave

Maude, estrechamiento, alcanzabilidad, lógica de reescritura, unificación, lógica ecuacional de pertenencia.

Abstract

This master's thesis presents a study about reachability problem solving in rewrite theories with an underlying membership equational logic, using the narrowing technique. To achieve this two calculi have been developed , one that solves the unification modulo equational logic problem and another one that solves the reachability problem based on the former unification calculus. Both calculi make use of membership, unification modulo axioms and matching algorithms, all of them available in the rewriting language Maude. Special emphasis has been made on both calculi in the efficient use of term typing information. Soundness and completeness of both calculi has been proved. Afterwards, two sets of transformation rules have been developed to allow the implementation of both calculi. Finally, those rules have been programmed on a prototype, using the rewriting language Maude.

Keywords

Maude, narrowing, reachability, rewriting logic, unification, membership equational logic.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

The aim of this work is to study the relationship between verifiable and computable answers to *reachability problems* in rewrite theories with an underlying membership equational logic. A reachability problem is an existential formula

$$(\exists \bar{x}) s(\bar{x}) \rightarrow^* t(\bar{x})$$

with $\bar{x}$ some variables, or a conjunction of several of these subgoals.

In this work, a calculus that solves this kind of problems has been developed for rewrite theories. Given a reachability problem in a rewrite theory, this calculus can compute any answer that can be checked by rewriting, or a more general one, one that subsumes the checked one. For instance, instead of $X \mapsto f(a, b, Z)$ where Z is a variable, a and b are constants, the calculus may find $X \mapsto f(Y, b, Z)$ where Y is also a variable. The calculus is first defined for equational unification modulo axioms and its correctness is proven. Then it is extended for reachability goals and also proven correct. The work is not concerned with proving termination in conditional rewriting (see [LMM05] for information on this subject). Special care has been taken in the calculus to keep membership information attached to each term, to make use of it whenever possible (for instance, dropping unfeasible goals or modifying the sort that a term must have depending on the sort of the other term it has to unify with). The calculus does not apply to generalized rewrite theories [BM06] having either *frozen* arguments [BM06] or context-sensitive strategies [CDE+]. We use the rewriting language Maude [CDE+02] as a tool for specifying rewrite theories and checking solutions for reachability problems. Some of the functions available on Maude, such as unification modulo axioms or matching modulo axioms, whose algorithms are very complex [HM12], are used in the calculus.

1.2 Motivation

Since the late 60's there has been a concern in the programming community about the semantics of programs. The increasing complexity of computer programs made necessary the development of languages, tools and mathematical methods that could improve the speed and safety when developing programs. One of the first milestones was Tony Hoare's axiomatic semantics [Hoa69]. In the following twenty years there were several proposals of languages, such as OBJ3 [GKK+87], and models for concurrent system specification, such as Petri nets [Pet73] or CCS [Mil80].

Rewriting logic is a computational logic that has been around for more than twenty years [Mes90], whose semantics [BM06] has a precise mathematical meaning allowing mathematical reasoning for property proving, as an attempt to provide a more flexible framework for the specification of concurrent systems. It turned out that it can express both concurrent computation and logical deduction, allowing its application in many areas such as automated deduction, software and hardware specification and verification, security, etc.

On the computational side, rewriting logic is a semantic framework that allows natural representation, execution and analysis as rewrite theories of different concurrency models, distributed algorithms, etc. On the logic side, it is a logical framework that allows representation and reasoning about different logics and automated decision procedures.

One important property of rewriting logic is that the distance between the represented structure and its representation in rewriting logic is very small. Usually they are isomorphic structures where differences are due to the notations used on both sides, but the main features remain the same. This allows reducing errors when coding.

Another important property of rewriting logic is reflection [CM96]. Intuitively, reflection means representing a logic's metalevel at the object level, allowing the definition of strategies that guide rule application in an object-level theory. A classic example of reflection can be found on Turing's Universal Machine [Tur36].

The reachability problem can be solved by model checking methods [CGP99] for finite state spaces. A technique known as *narrowing* [Fay78] that was first proposed as a method for solving equational goals (*unification*), has been extended to cover also reachability goals [DMT98], leaving equational goals as a special case of reachability goals. This technique resembles symbolic model checking, where we represent infinite sets of states using logical variables in terms. Variables get instantiated through the narrowing process, when necessary. In recent years the idea of *variants* of term [CLD05] has been applied to narrowing. The variants of a term s are pairs (t, θ), meaning that term s rewrites to the irreducible (*canonical*) term t using substitution θ. A strategy for order-sorted unconditional rewrite

theories known as *folding variant narrowing* [ESM12], which computes a complete set of variants of any term S, has been developed by Escobar, Sasse and Meseguer, allowing unification *modulo* a set of equations and axioms. The strategy terminates on any input term on those systems enjoying the *finite variant property* [CLD05], a characterization that ensures that any term has a finite number of variants, and it is *optimally terminating*, that is, if any complete narrowing strategy terminates on an input term, the folding variant narrowing will terminate on this term. It is being used for cryptographic protocol analysis [MT07], with tools like Maude-NPA [EMM05], termination algorithms modulo axioms [DLM09], and algorithms for checking confluence and coherence of rewrite theories modulo axioms, such as the Church-Rosser (CRC) and the Coherence (ChC) Checkers for Maude [DM12].

This work explores narrowing for membership conditional rewrite theories, going beyond the scope of folding variant narrowing which works on order-sorted unconditional rewrite theories. A calculus that computes answers to reachability problems in membership conditional rewrite theories has been developed and proved correct with respect to idempotent normalized answers, that is, given a solution for a reachability problem the calculus can compute one answer that subsumes (includes) this solution, and if the calculus computes one answer then the answer is a solution for the reachability problem.

1.3 Structure of the work

- In chapter 2 all needed definitions and properties for rewriting and narrowing are introduced.
- Chapter 3 is a brief introduction to the rewriting language Maude, emphasizing the needed parts of it.
- Chapter 4 introduces the first part of the narrowing calculus, the one that deals with equational unification. In this calculus the rule to apply each time is always correctly chosen (we have an oracle). All we show is that an answer exists (if it does). We are not concerned on how to choose rules (this is a strategy). There are several examples showing the calculus at work.
- In chapter 5 the proofs of soundness and completeness of the calculus for unification presented in chapter 4 are shown.
- In chapter 6 a set of transformations for the previous calculus is presented. An example shows the inner working of this set of transformations.
- Chapter 7 introduces the rest of the calculus, the part dealing with reachability. Again, we have an oracle that always guesses the right rule to apply.

- Chapter 8 holds the proofs of soundness and completeness of the calculus for reachability presented in chapter 7.
- In chapter 9 a set of transformations for the rest of the calculus is presented. Another example shows the inner working of this set of transformations. It is worth pointing that the whole set of transformations work at the metalevel, with the given rewrite theory as an object. This is where possible enhancements can be made via strategies.
- Chapter 10 discusses the implementation of the set of transformations, structure and flow control, together with improvements that can be made at this stage. Source code for the implementation, as well as several examples and instructions for its use can be found on http://maude.sip.ucm.es/cnarrowing/.
- In chapter 11, conclusions and further improvements and lines of investigation for this work are presented.

Chapter 2

Preliminaries

Rewriting logic, as it has been said, is a general logical framework in which many deductive systems can be naturally represented [BM06]. There are several language implementations of rewriting logic, including Maude [CDE+02]. Rewriting logic is parameterized by an underlying equational logic. In Maude's case this logic is membership equational logic [Mes97].

In this section we introduce the equational part of the logic, then the rewrite part of it. We follow by presenting sufficient conditions under which these logics are computable. Then *unification*, the problem of assigning values to variables in terms to make them equationally equal, is described. The equivalent problem for rewriting (*reachability*) is presented, and a technique (*narrowing*) that suits both problems is described. We end the section showing a transformation that turns a unification problem into a reachability one, allowing us to solve both kinds of problems using the same techniques.

Throughout this section a theoretical vending machine (what else!) will be used as a motivating example to explain the definitions in an less abstract way. This machine accepts a `Coin` (a quarter (`q`) or a dollar (`$`), as it is U.S. made) that may be inserted at any time, and nondeterministically serves one `Item` if there is enough credit: an apple (`a`) at a price of one dollar, or a coffee (`c`) at a price of three quarters. The vending machine is rather odd: in order to serve a coffee there must be a credit of at least a whole dollar; then the machine may serve the coffee (or an apple). The vending machine knows that four quarters make a dollar. If there is a credit of three quarters, the machine serves nothing (although it has enough money to serve a coffee). As there must always be a credit of a whole dollar before the vending machine serves anything, we never know if we are going to get a coffee or an apple. The vending machine has a `State` which is a nonempty multiset of `Coins` and `Items` (the initial `State` may not be empty). The `State` tells us the credit, and the `Items` that have already been served. A single `Coin` or `Item` is a `State`. `States` are written as a mere juxtaposition of `Coins` and `Items`,

that is, we use an *empty* operator. Parentheses may be used to enclose several items of a `State` if desired.

2.1 Membership equational logic

We first describe the static (equational) part of our theories. This includes the items we are going to work with (operators, terms, kinds, sorts, etc) as well as the criteria to consider that two syntactically different terms belong in the same class of terms (conditional equations and memberships), that is, we define equivalence classes on terms. We also define essential concepts, like positions or substitutions, which will be widely used.

A *membership equational logic* (MEL) *signature* [BM06] is a triple $\Sigma = (K, \Omega, S)$, with K a set of *kinds*, $\Omega = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded algebraic signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. The sets $T_{\Sigma,s}$, $T_\Sigma(X)_s$, $T_{\Sigma,k}$ and $T_\Sigma(X)_k$ denote, respectively, the set of ground Σ-terms with sort s, the set of Σ-terms with sort s over the set X of *sorted* variables, the set of ground Σ-terms with kind k and the set of Σ-terms with kind k over the set X of *sorted* variables. We write T_Σ, $T_\Sigma(X)$ for the corresponding term algebras. Given a term $t \in T_\Sigma(X)$, the set $vars(t) \subseteq X$ denotes the set of variables in t.

The MEL *signature* (Σ) for our vending machine has only one kind, $K = \{[\texttt{State}]\}$, with three sorts, $S_{[State]} = \{\texttt{State}, \texttt{Coin}, \texttt{Item}\}$. $S = \{S_{[\texttt{State}]}\}$. $\Omega = \{\cdot_{[\texttt{State}]\ [\texttt{State}],[\texttt{State}]}\}$, that is there is only one function ($\cdot$, understood as juxtaposition) that given a pair of elements with kind `[State]` returns another element with kind `[State]`. $T_{\Sigma,\texttt{Coin}} = \{\texttt{q}, \$\}$, $T_{\Sigma,\texttt{Item}} = \{\texttt{a}, \texttt{c}\}$. `q`, `$`, `a` and `c` are the *atoms* (or *atomic values*) of our signature. Any ground term has to be either one of this atoms or some term made up with this atoms and the only *constructor function* ($\cdot$).

When a term t is parsed as a tree, the empty string ϵ represents the root of t. *Positions* in a term t are denoted as strings of nonzero natural numbers and represent nodes or leaves of its parsed tree. The set of positions of a term is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. If we consider the subtree of t below a certain position p, p being the root of the subtree, we get a subterm of the term t denoted by $t|_p$. For instance the subterm at position 2 of $t \equiv f(a, g(b, c))$ is $t|_2 \equiv g(b, c)$. The replacement in t of a subterm at position p by another term u is denoted by $t[u]_p$.

A *substitution* $\sigma : Y \to T_\Sigma(X)$ is a function from a finite set of sorted variables $Y \subseteq X$ to $T_\Sigma(X)$ such that $\sigma(y)$ has the same or lower sort as that of the variable $y \in Y$ ($s_1 \leq s_2$, formally defined in the next paragraph). The application of a substitution σ to a term t is denoted by $t\sigma$. The substitution instance $t\sigma$ of a term

t is a term obtained from t by *simultaneously* replacing each occurrence of variable $y \in Dom(\sigma)$ in t with $\sigma(y)$. 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 $\sigma : Y \to T_\Sigma(X)$ are homomorphically extended to $T_\Sigma(X)$, written with the same notation $\sigma : T_\Sigma(X) \to T_\Sigma(X)$. For simplicity, we assume that every substitution is idempotent,that is, σ satisfies $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. The restriction of σ to a set of variables V is $\sigma|_V$; sometimes we write $\sigma|_{t_1,\ldots,t_n}$ to denote $\sigma|_V$ where $V = Var(t_1), \ldots, Var(t_n)$. Composition of two substitutions is denoted by $\sigma\sigma'$. Combination of two substitutions is denoted by $\sigma \cup \sigma'$. We call an idempotent substitution σ a variable renaming if there is another idempotent substitution σ^{-1} such that $(\sigma\sigma^{-1})|_{Dom(\sigma)} = id$.

In our vending machine, if $t = qX_{\texttt{Item}}$ and $\sigma = \{X_{\texttt{Item}} \mapsto c\}$ then $t\sigma = qc$, which is a term with sort `State`, as we will now see.

A MEL theory [BM06] is a pair $(\Sigma, \mathcal{E})$, where Σ is a MEL signature and $\mathcal{E}$ is a finite set of MEL sentences, either a conditional equation or a conditional membership of the forms:

$$(\forall X)\ t{=}t' \ \texttt{if}\ \bigwedge_i A_i, \quad (\forall X)\ t{:}s \ \texttt{if}\ \bigwedge_i A_i$$

for $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$, the latter stating that t is a term of sort s, provided the condition holds, and each A_i can be of the form $t{=}t'$, $t{:}s$ or $t{:=}t'$ (a *matching* equation). Matching equations are treated as ordinary equations, but they impose a limitation in the syntax of admissible MEL theories, as we will see. Order-sorted notation $s_1 \leq s_2$ can be used instead of $(\forall x{:}[s1])\ x{:}s_2$ `if` $x{:}s_1$. An operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1{:}k_1, \ldots, x_n{:}k_n)\ f(x_1, \ldots, x_n){:}s$ `if` $\bigwedge_{1 \leq i \leq n} x_i{:}s_i$. Given a MEL sentence ϕ, we denote by $\mathcal{E} \vdash \phi$ that ϕ can be deduced from $\mathcal{E}$ using the rules in Figure 2.1, where $=$ can be either $=$ or $:=$ as explained before [BM12].

The MEL theory for our vending machine consists of the MEL signature Σ defined before, and the following set $\mathcal{E}$ of MEL sentences:

- $\forall X{:}[\texttt{State}]\ X{:}\texttt{State}$ *if* $X{:}\texttt{Item}$ (every `Item` is a `State`, or `Item` $\leq$ `State`)
- $\forall X{:}[\texttt{State}]\ X{:}\texttt{State}$ *if* $X{:}\texttt{Coin}$ (every `Coin` is a `State`, or `Coin` $\leq$ `State`)
- $\forall X, Y{:}[\texttt{State}]\ XY{:}\texttt{State}$ *if* $X{:}\texttt{State} \wedge Y{:}\texttt{State}$
  (the juxtaposition of `States` is a `State`)
- $\forall X, Y{:}[\texttt{State}]\ XY = YX$ (juxtaposition is commutative)
- $\forall X, Y, Z{:}[\texttt{State}]\ (XY)Z = X(YZ)$ (juxtaposition is associative)

$$\frac{t \in T_\Sigma(X)}{(\forall X) t = t}\ \texttt{Reflexivity} \qquad \frac{(\forall X) t = t'}{(\forall X) t' = t}\ \texttt{Symmetry} \qquad \frac{(\forall X) t_1 = t_2 \quad (\forall X) t_2 = t_3}{(\forall X) t_1 = t_3}\ \texttt{Transitivity}$$

$$\frac{(\forall X) t' : s \quad (\forall X) t = t'}{(\forall X) t : s}\ \texttt{Membership} \qquad \frac{f \in \Sigma_{k_1 \cdots k_n, k} \quad (\forall X) t_i = t_i' \quad t_i, t_i' \in T_\Sigma(X)_{k_i}, 1 \leq i \leq n}{(\forall X) f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}\ \texttt{Congruence}$$

$$\frac{((\forall X)\ A_0\ \texttt{if}\ \bigwedge_i A_i) \in E \quad \theta : X \to T_\Sigma(Y) \quad (\forall Y) A_i \theta}{(\forall Y) A_0 \theta}\ \texttt{Replacement}$$

Figure 2.1: *Deduction rules for membership equational logic.*

- `qqqq` = \$ (four quarters make a dollar)

A Σ-*algebra* A [Mes97] consists of a set A_k for each kind k, a function $A_f : A_{k_1} \texttt{x} \cdots \texttt{x} A_{k_n}$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. For a valuation $a : X \to A$ assigning a value in A_s to each variable $x \in X$ with sort s, if $\bar{a} : T_\Sigma(X) \to A$ is the homomorphic extension of a to terms, by definition, $A, a \models (\forall X)\ t = t'$ iff $\bar{a}(t) = \bar{a}(t')$, and $A, a \models (\forall X)\ t : s$ iff $\bar{a}(t) \in A_s$. A Σ-*algebra* A is a model of a formula ϕ, written $A \models \phi$, when ϕ is satisfied for any valuation a. A MEL sentence φ is a logical consequence of $(\Sigma, \mathcal{E})$, written $(\Sigma, \mathcal{E}) \models \varphi$, when all the models of $(\Sigma, \mathcal{E})$ are also models of φ. The rules of Figure 2.1 specify a sound and complete calculus, that is, $(\Sigma, \mathcal{E}) \vdash \varphi \iff (\Sigma, \mathcal{E}) \models \varphi$. A MEL theory $(\Sigma, \mathcal{E})$ has an *initial algebra*, denoted by $T_{\Sigma/\mathcal{E}}$, whose elements are equivalence classes $[t]_\mathcal{E} \subseteq T_\Sigma$ of ground terms identified by the equations in $\mathcal{E}$.

The initial algebra for the vending machine is the set of all non-empty multisets that can be made up with the four atoms q, \$, c, a. Recall that, for instance $\{a, a, q\}$ and $\{a, q, a\}$ are the same multiset, but they are not the multiset $\{a, q\}$.

2.2 Rewriting logic

Now we describe the dynamic part of our theories. These are the conditional rewrite rules that make our system evolve, be it a concurrent or a deductive system.

A rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ is a formal specification of concurrent or deductive systems [Mes92], where

- $(\Sigma, \mathcal{E})$ is a theory in *membership equational logic*
- R is a finite set of *labeled conditional rewrite rules*, each of which has the form ($=$ can be either $=$ or $:=$):

$$\lambda : (\forall X)\ l \to r\ \texttt{if}\ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j \wedge \bigwedge_k l_k \to r_k,$$

  where l, r are Σ-terms of the same kind.

$$\frac{t \in T_\Sigma(X)}{(\forall X) t \to t}\ \texttt{Reflexivity}$$

$$\frac{(\forall X) t_1 \to t_2,\ (\forall X) t_2 \to t_3}{(\forall X) t_1 \to t_3}\ \texttt{Transitivity}$$

$$\frac{(\forall X) u \to u',\ \mathcal{E} \vdash (\forall X) t = u,\ \mathcal{E} \vdash (\forall X) u' = t'}{(\forall X) t \to t'}\ \texttt{Equality}$$

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k}\ (\forall X) t_i \to t_i'\ \ t_i, t_i' \in T_\Sigma(X)_{k_i}, 1 \leq i \leq n}{(\forall X) f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}\ \texttt{Congruence}$$

$$\frac{\begin{array}{c}(\lambda : (\forall X)\, l \to r \ \texttt{if} \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j \wedge \bigwedge_k l_k \to r_k) \in R \\ \theta : X \to T_\Sigma(Y) \quad \bigwedge_i \mathcal{E} \vdash (\forall Y) p_i\theta = q_i\theta \quad \bigwedge_j \mathcal{E} \vdash (\forall Y) w_j\theta : s_j \quad \bigwedge_k (\forall Y) l_k\theta \to r_k\theta\end{array}}{(\forall Y) l\theta \to r\theta}\ \texttt{Replacement}$$

Figure 2.2: *Deduction rules for rewrite theories.*

Such a rewrite rule specifies a *one-step transition* (often called a *one-step rewrite*) from a state $t[l\theta]_p$ containing a substitution instance $l\theta$ at a position p to the state $t[r\theta]_p$ in which $l\theta$ has been replaced by the corresponding instance $r\theta$, denoted by $t[l\theta]_p \to^1_R t[r\theta]_p$, provided the condition holds; that is, the substitution instance by θ of each condition in the rule follows from $\mathcal{R}$. The subterm $t|_p$ is called a *redex*.

In our vending machine, R is the following set of labeled conditional rewrite rules:

- add-quarter: $\forall X{:}[\texttt{State}]\ X \to Xq$ *if* $X{:}\texttt{State}$ (quarter inserted)
- add-dollar: $\forall X{:}[\texttt{State}]\ X \to X\$$ *if* $X{:}\texttt{State}$ (dollar inserted)
- buy-coffee: $\$ \to c$ (coffee served)
- buy-apple: $\$ \to aq$ (apple served, credit updated)

The inference rules [BM12] in Figure 2.2 for rewrite theories can infer all possible deductive computations in the system specified by $\mathcal{R}$. We can *reach* a state v from a state u if we can prove $\mathcal{R} \vdash u \to v$.

The relation $\to^1_{R/\mathcal{E}}$ on $T_\Sigma(X)$ is $=_\mathcal{E} \circ \to^1_R \circ =_\mathcal{E}$. $\to^1_{R/\mathcal{E}}$ on $T_\Sigma(X)$ induces a relation $\to^1_{R/\mathcal{E}}$ on $T_{\Sigma/\mathcal{E}}(X)$, the equivalence relation modulo $\mathcal{E}$, by $[t]_\mathcal{E} \to^1_{R/\mathcal{E}} [t']_\mathcal{E}$ iff $t \to^1_{R/\mathcal{E}} t'$. The transitive (resp. transitive and reflexive) closure of $\to^1_{R/\mathcal{E}}$ is denoted $\to^+_{R/\mathcal{E}}$ (resp. $\to^*_{R/\mathcal{E}}$). We say that a term t is $\to_{R/\mathcal{E}}$-*irreducible* (or just $R/\mathcal{E}$-irreducible) if there is no term t' such that $t \to^1_{R/\mathcal{E}} t'$.

We define now several properties that rewrite rules, rewrite theories substitutions or relations may have. It is not mandatory for all rewrite theories to have

all properties, but some of them will be required to make the rewrite theories computable.

For a rewrite rule $l \to r$ `if` *cond*, we say that it is *sort-decreasing* if for each substitution σ, we have that $r\sigma \in T_\Sigma(X)_s$ and $(cond)\sigma$ is verified implies $l\sigma \in T_\Sigma(X)_s$, that is, if we apply this rule to a term t with sort s, we get another t' whose sort s' is lower than or equal to s. We say that a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ is sort-decreasing if all rules in R are. For a Σ-equation $t = t'$, we say that it is *regular* if $Var(t) = Var(t')$, that is, there are no extra variables, and it is *sort-preserving* if for each substitution σ, we have $t\sigma \in T_\sigma(X)_s$ implies $t'\sigma \in T_\sigma(X)_s$ and vice versa. We say a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ is regular or sort-preserving if all equations in E are.

For substitutions σ, ρ and a set of variables V we define $\sigma|_V \to^1_{R/\mathcal{E}} \rho|_V$ if there is $x \in V$ such that $\sigma(x) \to^1_{R/\mathcal{E}} \rho(x)$ and for all other $y \in V$ we have $\sigma(y) =_\mathcal{E} \rho(y)$. A substitution is called $\mathcal{E}$-*normalized* (or normalized) if $x\sigma$ is $\mathcal{E}$-irreducible for all $x \in V$. This is the simplest version modulo $\mathcal{E}$ for that substitution.

We say that the relation $\to^1_{R/\mathcal{E}}$ is *terminating* if there is no infinite sequence $t_1 \to^1_{R/\mathcal{E}} t_2 \to^1_{R/\mathcal{E}} \cdots t_n \to^1_{R/\mathcal{E}} t_{n+1} \cdots$. We say that the relation $\to^1_{R/\mathcal{E}}$ is *confluent* if whenever $t \to^*_{R/\mathcal{E}} t'$ and $t \to^*_{R/\mathcal{E}} t''$, there exists a term t''' such that $t' \to^*_{R/\mathcal{E}} t'''$ and $t'' \to^*_{R/\mathcal{E}} t'''$. A rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ is confluent (resp. terminating) if the relation $\to^1_{R/\mathcal{E}}$ is confluent (resp. terminating). In a confluent, terminating, sort-decreasing, membership rewrite theory, for each term $t \in T_\Sigma(X)$, there is a unique (up to $\mathcal{E}$-equivalence) $R/\mathcal{E}$-irreducible term t' obtained by rewriting to *canonical* form, denoted by $t \to^!_{R/\mathcal{E}} t'$, or $t \downarrow_{R/\mathcal{E}}$ when t' is not relevant, which we call $can_{R/\mathcal{E}}(t)$. Then, we can apply any available rule each time, and obtain always the same canonical form modulo $\mathcal{E}$. We write $t \downarrow$ or $can(t)$ when the underlying rewriting logic is known.

Our vending machine is, as most reactive systems are [AILS07], non terminating. From any initial `State` we can always apply rules `add-quarter` and `add-dollar`. Rule `buy-coffee` is not sort-decreasing because it can turn a term with sort `Coin` into a term with sort `Item`, and it is not true that `Item` $\leq$ `Coin`. Also rule `buy-apple` is not sort-decreasing because it can turn a term with sort `Coin` into a term with sort `State`, which is strictly bigger than sort `Coin` (`Coin` $\leq$ `State` and `State` $\not\leq$ `Coin`.)

2.3 Executable rewrite theories

For a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, whether a one step rewrite $t \to^1_{R/\mathcal{E}} t'$ holds is undecidable in general. We impose additional conditions under which we can computationally decide if $t \to^1_{R/\mathcal{E}} t'$ holds. This conditions are not very restrictive

and, in fact, they allow a great number of systems to be specified. We decompose $\mathcal{E} = E \cup A$. A rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ is *executable* if each kind k in Σ is nonempty, E, A, and R are finite and the following conditions hold:

1. E and R are *admissible*, that is, the set E consists of conditional equations (1) and conditional memberships (2), and the set R consists of conditional rules (3), where in (1) the variables in t' are among those in t or in some A_i, and where, in (1), (2) and (3) each A_i can be either a membership $t_i{:}s_i$, equation $t_i{=}t'_i$ or matching equation $t_i{:=}t'_i$ such that any new variable not in t or in some A_j with $j < i$ must occur only in t_i or in some A_j with $j > i$; furthermore, if t_i introduces any new variables, then t_i must be a non variable term. In (3), given a conditional rule of the form

$$l : t \to t' \ \textit{if} \ A_1 \wedge \ldots \wedge A_n,$$

A_i can also be a rewrite $t_i \to t'_i$. Then it must satisfy the additional requirement

$$vars(t_i) \subseteq vars(t) \cup \bigcup_{j=1}^{i-1} vars(A_j),$$

and furthermore t'_i is an E-pattern[1]. Logically we treat matching equations as ordinary equations. The point with admissible theories is that they allow us to assign values to new variables by matching. With the conditions for a matching equation, its left side is an E-pattern, and can not be rewritten, so we rewrite the right side of the matching equation to canonical form and match it against the new variables in the left side.

2. Equality modulo A, i.e., $t =_A t'$, is decidable and there exists a *matching algorithm modulo* A, producing a finite number of A-matching substitutions or failing otherwise, that can implement rewriting in A-equivalence classes. Usually, A are axioms of commutativity, associativity and identity that may be non terminating under standard rewriting. We put this axioms apart from the terminating ones and use special algorithms for them, that are designed to avoid non terminating behaviors.

3. The equations E are *sort-decreasing*, and *terminating, coherent, and confluent modulo* A when we consider them as oriented rules. Sort-decreasingness,

[1] We call a term t an E-pattern if for any well-formed substitution σ such that for each variable x in its domain the term $\sigma(x)$ is in canonical form with respect to the equations in E, then $t\sigma$ is also in canonical form. A sufficient condition for t to be an E-pattern is the absence of unifiers (see 2.4) between its non variable subterms and left hand sides of equations in E. There is recent work on this kind of unification where one term is always in normal form so no rewrite rules can ever be applied to it, which has been called *asymmetric unification* [EEK+13]

confluence and termination allows us to represent $\mathcal{E}$-equivalence classes as A-equivalence classes in E/A-canonical form uniquely. The A-coherence assumption makes it possible to compute the rewrite relation $\rightarrow^1_{E/A}$ on A-equivalence classes by means of an A-matching algorithm.

4. The rules R are *coherent* relative to the equations E modulo A. That is, together with the above conditions, if t is rewritten to t' by a rule $(l \rightarrow r$ `if` $cond)$, the E-canonical term $can_{E/A}(t)$ is also rewritten to t'' by the *same* rule such that $can_{E/A}(t') =_A can_{E/A}(t'')$. Technically, what coherence means is that the weaker relation $\rightarrow^1_{E,A}$ becomes semantically equivalent to the stronger relation $\rightarrow^1_{E/A}$, so we can decide $t \rightarrow^1_{R/\mathcal{E}} t'$ by finding t'' such that $can_{E,A}(t) \rightarrow^1_R t''$ and $can_{E,A}(t') =_A can_{E,A}(t'')$, which is a decidable, since the number of rules is finite and A-matching is decidable.

The rewrite theory for our vending machine is executable if we decompose $\mathcal{E}$ in the following way: the set A has as elements the equations for the commutative and the associative properties for function $\cdot$ (juxtaposition), the set E has the other equation and all memberships. E and R are admissible because they are regular and don't introduce new variables. A has a matching algorithm (when we use Maude it is called *match*). The equations E are *sort-decreasing*, and *terminating, coherent, and confluent modulo* A when we consider them as oriented rules and the rules R are *coherent* relative to the equations E modulo A. We will not go further into these properties, that must be checked by the user, but Maude provides tools like the Church-Rosser (CRC) and the Coherence (ChC) Checkers for Maude [DM12] that help the user verify them.

For executable rewrite theories $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ with $\mathcal{E} = E \cup A$ we define the relation $\rightarrow^1_{E,A}$ on $T_\Sigma(X)$ as follows: $t \rightarrow^1_{E,A} t'$ if there is an $\omega \in Pos(t)$, $l = r \in E$, and a substitution σ such that $t|_w =_A l\sigma$ (A-matching) and $t' = t[r\sigma]_\omega$. Since A is sort-preserving and E is sort-decreasing, t' is well-sorted, that is $t \in T_\Sigma(X)_s$ implies $t' \in T_\Sigma(X)_s$. The relation $\rightarrow^1_{R,A}$ is similarly defined, and because of our assumption about the signature Σ, it is the case that $t \rightarrow^1_{R,A} t'$ implies t' is well-sorted, and $t \in T_\Sigma(X)_{[s]}$ implies $t' \in T_\Sigma(X)_{[s]}$. We define $\rightarrow^1_{R\cup E,A}$ as $\rightarrow^1_{R,A} \cup \rightarrow^1_{E,A}$. Note that, since A-matching is decidable, $\rightarrow^1_{E,A}$, $\rightarrow^1_{R,A}$, and $\rightarrow^1_{R\cup E,A}$ are decidable. These three relations are lifted to substitutions as expected. $R \cup E$, A-normalized (and similarly R, A or E, A-normalized) substitutions are defined as expected.

2.4 Unification

Unification tries to assign values to variables in two terms t and t' through a substitution σ in a way such that they are semantically equal ($t\sigma =_\mathcal{E} t'\sigma$) [Baa90]. In

membership equational logic we answer the question $\forall(\bar{x})t(\bar{x})=t'(\bar{x})$? In unification we answer the question $\exists(\bar{x})t(\bar{x})=t'(\bar{x})$?

For instance, the membership equational logic for the vending machine tells us that if X is a variable with sort `State` then $Xqqqq = X\$$ whatever X is, but with unification we know that $Xqqq = \$$ only if we use the substitution $\{X \mapsto q\}$.

Given an executable rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, a Σ-*equation* is an expression of the form $t = t'$ where $t, t' \in T_\Sigma(X)_s$ for an appropriate s. The $\mathcal{E}$-*subsumption* preorder $\ll_\mathcal{E}$ on $T_\Sigma(X)_s$ is defined by $t \ll_\mathcal{E} t'$ (meaning that t' is more general than t) if there is a substitution σ such that $t =_\mathcal{E} t'\sigma$; such a substitution σ is said to be an $\mathcal{E}$-*match* from t to t'. For substitutions σ, ρ and a set of variables V we define $\sigma|_V =_\mathcal{E} \rho|_V$ if $\sigma(x) =_\mathcal{E} \rho(x)$ for all $x \in V$, and $\sigma|_V \ll_\mathcal{E} \rho|_V$ if there is a substitution η such that $\sigma|_V =_\mathcal{E} (\rho\eta)|_V$ (we say that ρ is more general than σ).

A *system of equations* F is a conjunction of the form $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$ where for $1 \leq i \leq n$, $t_i = t'_i$ is a Σ-equation. We define $Var(F) = \bigcup_i Var(t_i) \cup Var(t'_i)$. An $\mathcal{E}$-*unifier* for F is a substitution σ such that $t_i\sigma =_\mathcal{E} t'_i\sigma$ for $1 \leq i \leq n$. When $\mathcal{E} = \emptyset$ (no associativity, no commutativity, etc) there is at much one unifier. In the general case, the set of unifiers for a system of equations may not be finite. For $V = Var(F) \subseteq W$, a set of substitutions $CSU_\mathcal{E}(F, W)$ is said to be a *complete* set of unifiers of F away from W [GS89] if

- each $\sigma \in CSU_\mathcal{E}(F, W)$ is an $\mathcal{E}$-unifier of F;
- for any $\mathcal{E}$-unifier ρ of F there is a $\sigma \in CSU_\mathcal{E}(F, W)$ such that $\rho|_V \ll_\mathcal{E} \sigma|_V$;
- for all $\sigma \in CSU_\mathcal{E}(F, W)$, $Dom(\sigma) \subseteq V$ and $Ran(\sigma) \cap W = \emptyset$.

That is, a complete set of unifiers $CSU_\mathcal{E}(F, W)$ is composed of idempotent $\mathcal{E}$-unifiers of F such that they only instantiate variables on F, no new variable on the unifiers belongs to the set W, and for any other $\mathcal{E}$-unifier of F there is a more general one in $CSU_\mathcal{E}(F, W)$ with respect to the variables in F.

An $\mathcal{E}$-unification algorithm is *complete* if for any given system of equations it generates a complete set of $\mathcal{E}$-unifiers, which may not be finite. A unification algorithm is said to be *finite* and complete if it terminates after generating a finite and complete set of solutions.

Checking if ρ is an $\mathcal{E}$-unifier of F is achieved by E, A-rewriting. Using the equations in E as oriented rules and the matching algorithm for A we rewrite the terms in F to canonical form and check if each left side canonical term $can_{E/A}(t_i\rho)$ is equal modulo A (we use the matching algorithm) to the corresponding right side canonical term $can_{E/A}(t'_i\rho)$.

2.5 Reachability goals

Reachability goals and their solving are the main subjects of this work. We first define reachability goals, then we define solutions and trivial solutions of them. We follow by characterizing the needed properties in our rewrite theories that makes us able to compute solutions of reachability problems.

Given a rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$, a *reachability goal* G is a conjunction of the form $t_1 \rightarrow^* t'_1 \wedge \ldots \wedge t_n \rightarrow^* t'_n$ where for $1 \leq i \leq n$, $t_i, t'_i \in T_\Sigma(X)_{s_i}$ for appropriate s_i. We say that t_i are the *sources* of the goal G, while t'_i are the *targets*. We define $Var(G) = \bigcup_i Var(t_i) \cup Var(t'_i)$. A substitution σ is a *solution* of G if $t_i\sigma \rightarrow^*_{R/\mathcal{E}} t'_i\sigma$ for $1 \leq i \leq n$. We define $\mathrm{E}(G)$ to be the system of equations $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$. We say σ is a *trivial solution* of G if it is an $\mathcal{E}$-unifier for $\mathrm{E}(G)$. We say G is trivial if the identity substitution id is a trivial solution of G.

For instance, in the rewrite theory for the vending machine if X is a variable with sort `State`, then $\{X \mapsto q\}$ is a trivial solution of the reachability goal $G \equiv Xqqq \rightarrow \$$, but it is a non-trivial solution of the reachability goal $G \equiv Xqq \rightarrow \$$ ($qqq \rightarrow_{\texttt{add-quarter}} qqqq \rightarrow_{equality} \$$).

For goals $G : t_1 \rightarrow^* t_2 \wedge \ldots \wedge t_{2n-1} \rightarrow^* t_{2n}$ and $G' : t'_1 \rightarrow^* t'_2 \wedge \ldots \wedge t'_{2n-1} \rightarrow^* t'_{2n}$ we say $G =_\mathcal{E} G'$ if $t_i =_\mathcal{E} t'_i$ for $1 \leq i \leq 2n$. We say $G \rightarrow_R G'$ if there is an odd i such that $t_i \rightarrow_R t'_i$ and for all $j \neq i$ we have $t_j = t'_j$. That is, G and G' differ only in one subgoal ($t_i \rightarrow t_{i+1}$ vs $t'_i \rightarrow t_{i+1}$), but $t_i \rightarrow t'_i$, so when we rewrite t_i in G to t'_i we get G'. We write $G \rightarrow_{r,R} G'$ meaning that rule $r \in R$ has been used in the rewriting step from G to G'. The relation $\rightarrow_{R/\mathcal{E}}$ over goals is defined as $=_\mathcal{E} \circ \rightarrow_R \circ =_\mathcal{E}$.

We implement $\rightarrow_{R/\mathcal{E}}$ (on terms and goals) using $\rightarrow_{R \cup E,A}$ [MT07]. This lemma links $\rightarrow_{R/\mathcal{E}}$ with $\rightarrow_{E,A}$ and $\rightarrow_{R,A}$. Patrick Viry gave a proof for unsorted unconditional rewrite theories [Vir94], which can easily be applied to our membership conditional case [MT07].

Lemma Let $\mathcal{R} = (\Sigma, \mathcal{E}, R)$ be an executable rewrite theory, that is, it has all the properties specified in section 2.3. Then $t_1 \rightarrow_{R/\mathcal{E}} t_2$ if and only if $t_1 \rightarrow^*_{E,A} \rightarrow_{R,A} t_3$ for some $t_3 =_\mathcal{E} t_2$.

Then $t_1 \rightarrow_{R/\mathcal{E}} t_2$ if and only if $t_1 \rightarrow^*_{E,A} \rightarrow_{R,A} t_3$ for some $t_3 =_\mathcal{E} t_2$. Thus $t_1 \rightarrow^*_{R/\mathcal{E}} t_2$ if and only if $t_1 \rightarrow^*_{R \cup E,A} t_3$ for some $t_3 =_\mathcal{E} t_2$, which can be decided by checking $t_3{\downarrow_{E,A}} =_A t_2{\downarrow_{E,A}}$ with the A-matching algorithm. This is the way rewriting is decided: from term t_1 we compute its derivation tree in a breadth-first way and check each resulting term against t_2 with the A-matching algorithm. If some term matches then the rewriting is possible and we have found a proof for it. This result is lifted to goals as $G_1 \rightarrow^*_{R/\mathcal{E}} G_2$ if and only if $G_1 \rightarrow^*_{R \cup E,A} G_3$ for some $G_3 =_\mathcal{E} G_2$. Also, σ is a trivial solution of $t_1{=}t'_1 \wedge \ldots \wedge t_n{=}t'_n$ if and only if

$t_1\sigma{\downarrow_{E,A}} =_A t'_1\sigma{\downarrow_{E,A}} \wedge \ldots \wedge t_n\sigma{\downarrow_{E,A}} =_A t'_n\sigma{\downarrow_{E,A}}$.

2.6 Narrowing

Narrowing is like rewriting, but replacing matching modulo an equational theory with unification modulo that theory. It tries to assign values to variables in two terms t and t' through a substitution σ in a way such that $t\sigma \rightarrow_{R/\mathcal{E}} t'\sigma$ (see [KKK+96, Ch. 14, p. 181-190] for a full description). In rewriting logic we answer the question $\forall(\bar{x})t(\bar{x}) \rightarrow t'(\bar{x})$? In narrowing we answer the question $\exists(\bar{x})t(\bar{x}) \rightarrow t'(\bar{x})$?. Unification is the only allowed way to assigns values, ground or not, to variables in narrowing. We don't guess values, we unify two terms modulo the given equational theory and use the resulting substitution as a partial or total answer.

In the vending machine example we can prove by rewriting that for a variable X with sort `State` $X\$ \rightarrow Xc$ whatever value X is given. Finding out that the substitution $\{X \mapsto q\}$ is a solution of the reachability goal $Xqqq \rightarrow c$ requires narrowing.

Let t be a Σ-term and W be a set of variables such that $Var(t) \subseteq W$. The R, A-*narrowing* relation on $T_\Sigma(X)$ is defined as follows: $t \rightsquigarrow_{p,\sigma,R,A} t'$ if there is a non-variable position $p \in Pos_\Sigma(t)$, a rule $l \rightarrow r$ `if` $cond$ in R, properly renamed, such that $Var(l) \cap W = \emptyset$, and a unifier $\sigma \in CSU_A^{W'}(t|_p = l)$ for $W' = W \cup Var(l)$, such that $t' = (t[r]_p)\sigma$ and $(cond)\sigma$ holds. This is lifted to reachability goals as follows. Let $G : t_1 \rightarrow^* t_2 \wedge \ldots \wedge t_{2n-1} \rightarrow^* t_{2n}$ and $G' : t'_1 \rightarrow^* t'_2 \wedge \ldots \wedge t'_{2n-1} \rightarrow^* t'_{2n}$, and suppose that $Var(G) \subseteq V$. We define $G \rightsquigarrow_{\sigma,R,A} G'$ if there is an *odd* i such that $t_i \rightsquigarrow_{p,\sigma,R,A} t'_i$ for some σ that is away from $Var(G)$, and for all $j \neq i$ we have $t'_j = t_j\sigma$. We write $G \rightsquigarrow^*_{\sigma,R,A} G'$ if either $G = G'$ and $\sigma = id$, or there is a sequence of derivations $G \rightsquigarrow_{\sigma_1,R,A} \ldots \rightsquigarrow_{\sigma_n,R,A} G'$ such that $\sigma = \sigma_1 \ldots \sigma_n$. Similarly E, A-narrowing and $R \cup E, A$-narrowing relations are defined on terms and goals, as expected.

Back to our vending machine and the reachability goal $Xqqq \rightarrow c$, we have that $(Xqqq)\sigma \equiv qqqq \rightsquigarrow_{\epsilon,\sigma,R\cup E,A} \$ \equiv (\$)\sigma$ using the oriented equation `qqqq` $= \$$ as a rule and unifier $\sigma{=}\{X \mapsto q\} \in CSU_A^{W'}(Xqqq|_\epsilon = qqqq)$, and $\$ \rightsquigarrow_{\epsilon,id,R\cup E,A} c$ using rule buy-coffee: $\$ \rightarrow c$, so it takes two narrowing steps, one with an oriented equation and another one with a rule, to find the answer.

2.7 Unification by rewriting

We have defined unification, but we have not given a method to compute unifiers. In this section we show an equivalent definition for executable Mel theories that

makes this computation possible. Furthermore, our calculus will make use of this equivalence to intermix both rewritings, for unification and reachability, instead of carrying an independent computation with each one.

2.7.1 Associated rewrite theory

Any executable MEL theory $(\Sigma, E \cup A)$ has a corresponding rewrite theory $\mathcal{R}_E = (\Sigma', A \cup M_E, R_E)$ associated to it, defined in [DLM+08, Ch. 3, p. 10-13], that allows us to check if a substitution σ is a solution for a goal G through rewriting instead of equational unification. We will use either of this approaches when proving properties of the calculus. It is defined in [DLM+08] as follows: we add a fresh new kind *Truth* with a constant tt to Σ, and for each kind $k \in K$ an operator $eq : k\ k \to Truth$. We write $\top$ to represent a conjunction of any number of tt's. The equational axioms are the ones in A. There are rules $eq(x{:}k, x{:}k) \to tt$ for each kind $k \in K$. Furthermore, for each admissible conditional equation in E the set R_E has a conditional rule of the form

$$t \to t' \ \textit{if}\ A_1^\bullet \wedge \ldots \wedge A_n^\bullet$$

where if A_i is a membership then $A_i^\bullet{=}A_i$, if A_i is a matching equation $t_i{:=}t_i'$ then $A_i^\bullet$ is the rewrite condition $t_i'{\to}t_i$, and if A_i is an ordinary equation $t{=}t'$ then $A_i^\bullet$ is the rewrite condition $eq(t,t'){\to}tt$. Similarly, for each conditional membership in E we add to M_E a conditional membership, with $A_i^\bullet$ as before, of the form,

$$t{:}s \ \textit{if}\ A_1^\bullet \wedge \ldots \wedge A_n^\bullet$$

Systems of equations in $(\Sigma, E \cup A)$ with form $G \equiv \bigwedge_{i=1}^m (s_i = t_i)$ become reachability goals in $\mathcal{R}_E$ with form $\bigwedge_{i=1}^m eq(s_i, t_i) \to tt$. A substitution σ is a solution of G if there are derivations for $\bigwedge_{i=1}^m (s_i\sigma = t_i\sigma)$, or $\bigwedge_{i=1}^m eq(s_i\sigma, t_i\sigma)$ rewrites to $\top$.

The inference rules for membership rewriting in $\mathcal{R}_E$ are the ones in Figure 2.3, adapted from [DLM+08, Fig. 4, p. 12], where the rules are defined for context-sensitive membership rewriting.

2.7.2 Computing $\mathcal{E}$-unifiers

Replacing in the inference rules for replacement and membership the matching condition $u =_A t\sigma$ with the unification condition $u\sigma =_A t\sigma$ will allow the forthcoming calculus for unification to compute the answer to unification problems modulo $\mathcal{E}$ using a unification algorithm modulo A and rewriting.

- (Reflexivity)

$$\frac{}{t \to t'}$$

if $t =_A t'$

- (Transitivity)

$$\frac{t_1 \to t_2,\ t_2 \to t_3}{t_1 \to t_3}$$

- (Congruence)

$$\frac{t_i \to t_i'}{f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, t_i', \ldots, t_n)}$$

- (Replacement)

$$\frac{A_1^\bullet \sigma \ldots A_n^\bullet \sigma}{u \to t'\sigma}$$

if $t \to t'$ *if* $A_1^\bullet \ldots A_n^\bullet$ in $\mathcal{R}_E$ and $u =_A t\sigma$

- (Subject Reduction)

$$\frac{t \to t', t' : s}{t : s}$$

- (Membership)

$$\frac{A_1^\bullet \sigma \ldots A_n^\bullet \sigma}{u : s}$$

if $t : s$ *if* $A_1^\bullet \ldots A_n^\bullet$ in $\mathcal{R}_E$ and $u =_A t\sigma$

Figure 2.3: *Inference rules for membership rewriting.*

Chapter 3

Maude

Maude is a high-level language and high-performance system supporting both equational and rewriting computation [CDE+02]. Maude's underlying equational logic is membership equational logic, which is an improvement over order-sorted algebra, allowing the faithful specification of types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties [BM06]. Maude has two kinds of *modules* that are of interest for our purpose:

- *Functional modules* provide support for functional programming in membership equational logic.
- *System modules* allow the specification of concurrent systems, when used as a semantic framework, or deductive systems, when used as a logical framework, using rewriting logic.

Moreover, Maude makes a systematic and efficient use of reflection, where programs are represented as data, allowing metaprogramming and metalanguage applications, as well as extensions to the language itself. In fact, standard Maude is known as Core Maude and there is an extension known as Full Maude [Dur99] programmed in Maude, where all new characteristics of the system are developed and tested prior to their inclusion in Core Maude. Among other characteristics, Full Maude offers support for object oriented modules and parameterized modules.

3.1 Functional modules

Maude's functional modules allow the specification and execution of MEL theories $(\Sigma, E \cup A)$, where A is a set of equational axioms (usually commutativity, associativity and/or identity) for some of the operators in the signature, and E is a

set of equations that are valid modulo A, as long as they are executable in the sense defined in section 2.3, that is we can always rewrite a term t to its canonical form $can_{E/A}(t)$ just applying equations in E as oriented rules, modulo A with the existing matching algorithm, until no more equations can be applied, being this always a finite process.

We show the syntax of functional modules through an example where we define natural numbers with an operation (sum), and sets of natural numbers:

```
fmod NAT-SET is
sorts Nat Set .
subsort Nat < Set .
op 0 : -> Nat .
op s : Nat -> Nat .
op _+_ : Nat Nat -> Nat .
op empty-set : -> Set .
op __ : Set Set -> Set [assoc comm id: empty-set] .
vars N M : Nat .
eq 0 + N = N .
eq s(N) + M = s(N + M) .
eq N N = N .
endfm
```

We can use Maude's command `reduce` to compute the canonical form of any term. For instance:

```
Maude> reduce 0 s(0) s(0) .
result Set: 0 s(0)
```

We declare sorts using the reserved word `sort`. Kinds are not defined in an explicit way. We refer to the kind of a sort s as $[s]$. Sort ordering, which as we saw in section 2.1 is a shortcut for certain membership axioms, is defined using the reserved word `subsort`.

Functions are declared using the reserved word `op` followed by the name of the function (which can be empty), the sort of the arguments and the sort of the result. The position of the arguments is determined by the "_" symbol that appears in the definition. The symbol -> separates the input arguments from the result. If no sort is found to the left of ->, the function is a constant. If no "_" symbol appears, the standard syntax for functions, with the arguments surrounded by brackets, is used. Axioms from A and other properties of the function are declared writing them between square brackets. In our example, the set constructor definition, which has empty name, has associative (`assoc`) and commutative (`comm`) properties, as expected for a set, and the identity element (`id`) is the empty set (`empty-set`).

Uniqueness of the elements in the set is guaranteed by the equation `eq N N = N` which deletes possible duplicated elements.

Variables are declared using the reserved word `var`. They can also be used without previous declaration by writing their name, a colon and its sort (for instance `N:Nat`).

Equational axioms are declared using the reserved words `eq`, or `ceq` when declaring conditional equational axioms. Similarly, membership axioms are declared using the reserved words `mb` and `cmb`.

Maude does not check confluence and termination properties for functional modules: the user is responsible for providing them. However, in some cases it is possible to check these properties with Maude's Church-Rosser checker and termination tools [DM10] [DLM+08].

3.2 System modules

System modules are an extension of functional modules that allow the specification of rewrite rules. The equational part must have the same properties as for functional modules. Rules are only required to be admissible in the sense defined in section 2.3.

The same syntax is used with the exception that the module begins with the reserved word `mod`, it ends with the reserved word `endm`, and rewriting rules are declared using the reserved words `rl`, or `crl` for conditional rewriting rules. Our module could be extended, for instance, with one rule that computes the sum of the terms in a set, becoming:

```
mod NAT-SET is
sorts Nat Set .
subsort Nat < Set .
op 0 : -> Nat .
op s : Nat -> Nat .
op _+_ : Nat Nat -> Nat .
op empty-set : -> Set .
op __ : Set Set -> Set [assoc comm id: empty-set] .
vars N M : Nat .
var S : Set
eq 0 + N = N .
eq s(N) + M = s(N + M) .
eq N N = N .
rl N M S => (N + M) S .
endm
```

Now we can use Maude's command `rewrite` to rewrite any term with the existing rules:

```
Maude> rewrite s(0) s(s(0)) s(0) .
result Nat: s(s(s(0)))
```

Prior to rewriting, Maude always reduces terms to canonical form. That's why the answer is `s(s(s(0)))` instead of `s(s(s(s(0))))`. We have given Maude a multiset, and applying rule `eq N N = N`, Maude deletes the spare `s(0)`. Then it applies the rule (matching `S` to `empty-set`) and returns the answer (also reduced, because with the previous matching we get `s(s(s(0))) empty-set` as answer, but since `empty-set` is declared as the identity element in `op __`, Maude applies this axiom and returns us `s(s(s(0)))`). In this case the answer is unique but there can be multiple answers, in which case Maude returns us only one, or none if the system is non terminating.

3.3 The metalevel

Maude reflective capacities are supported through the functional module META-LEVEL where each element we have defined previously has a correspondent sort (`Fmodule, Module, Term, ...`). This module is included in the file `prelude.maude` which is always loaded at the beginning of a session in Maude providing several often needed modules. One important sort in META-LEVEL is `Qid` (for quoted identifier), which allows us to meta-represent constants and variables by their own names preceded by an apostrophe (') and followed by a colon and a sort in the case of variables, or by a period and a sort in the case of constants (for instance `'N:Nat '0.Nat` in our previous example).

The META-LEVEL module provides several functions (`metaUnify, glbSorts, leastSort, ...`) that will be used in the calculus. These functions are only available at the metalevel because they must take the given theory as a parameter. For instance, the unification algorithm is *theory-dependent*, since a different order-sorted unification algorithm is derived for each different signature Σ and combination of axioms A, so the `metaUnify` command needs both as parameters (it really takes as only parameter the metarepresentation of the full module provided by the metalevel operation `upModule`).

A full coverage of Maude and its metalevel can be found on *Maude manual* [CDE+].

Chapter 4

Conditional narrowing modulo unification

Narrowing allows us to find possible values for variables in a way such that a reachability goal holds. Our intention is to partially emulate narrowing using a calculus that has the following properties:

1. If σ is a normalized idempotent solution for a reachability goal G, the calculus can compute a more general answer $\sigma \ll_{\mathcal{E}} \sigma'$ for G.

2. If the calculus computes an answer σ for G, then σ is a solution for G.

That is, we want to compute a complete set of answers for G, a set that includes a generalization of any possible solution for G.

We are going to split this task into two subtasks: first we will see the part of the calculus that deals with unification; second, we will see the part that deals with reachability.

4.1 Calculus rules for unification

We assume we are working with a Maude module named M. This module has all the declarations for sorts, kinds, operators, memberships, equations, axioms and rules.The calculus will make use of several functions at the metalevel, provided by Maude. Their syntax is simplified for clarity:

- $acuCohComplete(M)$, returns an ACU-coherence completed version of M[1].

[1]ACU-coherence completion [JKK83] guarantees that an equation or rule can be applied anywhere in associative-commutative functions, by adding extra equations. For instance, if we have the term $a + (b + c)$, where $+$ is associative-commutative, and the equation $a + c = d$, ACU-coherence completion adds an equation $a + c + X = d + X$, whose left part unifies with $a + (b + c)$, using substitution $\{X \mapsto b\}$, so we can rewrite $a + (b + c)$ to $d + b$.

- *glbSorts*(M, S, T) returns a set of sorts that are the *greatest lower bound* of sort S and sort T according to M. That is, if $R \in$ *glbSorts*(M, S, T), then $R \leq S$, $R \leq T$ and there is no R' such that $R \leq R' \leq S$ and $R \leq R' \leq T$ with the memberships in M.

- *unify*(M, s, t, n) returns the n-th A-unifier for s and t, that is, a substitution σ such that $s\sigma =_A t\sigma$, if it exists. (Actually, we will use its metalevel counterpart *metaUnify*.)

- *reduce*(M, s) returns a pair whose first element is term s reduced, and the second element is the sort for this reduced term. (Again, we will use its metalevel counterpart *metaReduce*.)

- *leastSort*(M, t) returns the least sort that term t can have without rewriting it, that is, only looking at the sort of its subterms, the definition of its operators and the memberships in M.

- *getType*(X) returns the sort of the variable X.

Maude's function *unify* is guaranteed to return a complete set of order-sorted unifiers, but it doesn't work with memberships. To overcome this problem we will use an approach similar to [Rie12]. We use the functions at the kind level, that is, all variables in terms are replaced by variables with the same name whose sort is the kind of the replaced variable, checking the memberships for these variables separately. In this way, axiom information is taken into account for A-unification, but membership information is not, so we will usually get a larger number of unifiers. The spurious ones will be later deleted by membership checking. For example, if we have to *unify* $f(X{:}S, a)$ and $g(b, Y{:}T)$ we *unify* the terms $f(X{:}[S], a)$ and $g(b, Y{:}[T])$, take each returned A-unifier and check that $X{:}[S]$ has sort S or $Y{:}[T]$ has sort T if any of them have been assigned a value in the A-unifier. The A-unifiers that don't pass this check are discarded. If any of the terms to unify is a variable $X{:}S$ we don't have to include any additional checking for its sort S, because it already has to be checked by the calculus for unification.

We complete the module M in the following way:

- For each operator $f : S_1 \dots S_n \to S$, $n \geq 0$, we add the membership *mb* $f(X_1{:}S_1, ..., X_n{:}S_n) : S$ to M, translating implicit sort and operator membership information into explicit memberships.

- We call *acuCohComplete*(M) to obtain an ACU-coherence completed version of M.

We will refer to the completed set of equations and memberships as E, to the completed set of rules as R and to the set of axioms as A.

A unification equation is a term $s{:}S = t{:}T$. This means that we intend to unify s and t, with resulting sorts S and T respectively, that is, we want to find a substitution σ such that $s\sigma$ has sort S, $t\sigma$ has sort T and σ is an $\mathcal{E}$-unifier for s and t. A unification goal is a sequence (understood as conjunction) of unification equations.

Admissible goals, or simply goals, are any sequence of $s{:}S{=}t{:}T$, $s{:}S{\approx}t{:}T$, $s{:}S{:=}t{:}T$ and $t{:}T$. From a unification goal the calculus tries to derive the empty goal. This part of the calculus is based on the inference rules for membership rewriting. In rewriting we work at the kind level, but any goal in our calculus of the form $s{:}S$ *op* $t{:}T$ is equivalent to the system of equations s *op* t, $s{=}X_S$, $t{=}Y_T$, that is s and t can unify at the kind level, but each one must unify with a variable of the required sorts, and then by membership they must also have that sorts (we will extend the syntax for systems of equations and allow the use of the equivalent requirements $s{:}S$, $t{:}T$.)

We use in our calculus a symbol $\approx$, not present in Σ, that only appears in root positions of terms. This symbol means rewriting using oriented equations as rules. We use it to distinguish between rewriting with oriented equations and rewriting with rules, where we will use the symbol $\rightarrow$.

Conditions in equations and memberships may have the form s or $t == t'$, where s is a term with sort *boolean*, which is a predefined sort in Maude. The predicate $==$ is a built-in boolean predicate of Maude that checks for syntactic equality. Given two terms, it reduces both to canonical form and checks if both are exactly the same. If this is the case, it rewrites the predicate to the boolean value *true*. Otherwise, it rewrites the predicate to the boolean value *false*. we will translate this conditions to $s \approx \mathit{true}$ or $t == t' \approx \mathit{true}$ respectively.

Our calculus is defined by the following set of inference rules, based on the concepts of *equational conditional rewriting without evaluation of the premise* [Boc93] and *lazy conditional narrowing calculus* [MSH02], where we assume that we have a numerable set of fresh variables (variables not present in any of the goals) for each sort. If we have to *unify* two terms s and t, we will call s', t' the kinded variable terms and c' the check for memberships generated by the transformation of s and t. Notice also that the first two rules, $[u]$ and $[x]$, transform *equational* problems into *rewriting* problems modulo axioms:

$[u]$ *unification*

$$\frac{G', s{:}S{=}t{:}T, G''}{G', s{:}S'{\approx}X{:}S', t{:}S'{\approx}X{:}S', G''}$$

where X fresh variable, $S' \in \mathit{glbSorts}(M, S, T)$.

[x] *matching*

$$\frac{G', s{:}S := t{:}T, G''}{G', t{:}S' \approx s{:}S', G''}$$

where $S' \in \mathit{glbSorts}(M, S, T)$.

[n] *narrowing*

$$\frac{G', s{:}S \approx t{:}T, G''}{(G', (c',)s{:}S', (c,)r{:}S' \approx t{:}S', G'')\theta}$$

(c)eq $l = r$ $(\mathit{if}\ c) \in E$ has fresh variables, $S' \in \mathit{glbSorts}(M, S, T)$,
θ A-unifier of s' and l' kinded variable terms, c' membership checks.

[t] *transitivity*

$$\frac{G', s{:}S \approx t{:}T, G''}{G', s{:}S' \approx X{:}S', X{:}S' \approx t{:}S', G''}$$

where X fresh variable, $S' \in \mathit{glbSorts}(M, S, T)$.

[i] *imitation*

$$\frac{G', f(\bar{s}{:}\bar{S}){:}S \approx X{:}T, G''}{G'\theta, s_i{:}S_i \approx X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S', f(\bar{X}{:}\bar{S}){:}S', G''\theta}$$

where $X \notin \mathit{Var}(s), \theta = \{X \mapsto f(\bar{X}{:}\bar{S})\}$,
X_i fresh variables, $S' \in \mathit{glbSorts}(M, S, T)$.

[d] *decomposition*

$$\frac{G', f(\bar{s}{:}\bar{S}){:}S \approx f(\bar{t}{:}\bar{T}){:}T, G''}{G', s_1{:}S_1 \approx t_1{:}T_1, ..., s_n{:}S_n \approx t_n{:}T_n, f(\bar{s}{:}\bar{S}){:}S', f(\bar{t}{:}\bar{T}){:}S', G''}$$

where $S' \in \mathit{glbSorts}(M, S, T)$.

[r] *removal of equations*

$$\frac{G', s{:}S \approx t{:}T, G''}{(G', (c',)s{:}S', t{:}S', G'')\theta}$$

where θ A-unifier of s' and t' kinded variable terms,
c' membership checks, $S' \in \mathit{glbSorts}(M, S, T)$.

[s] *subject reduction*

$$\frac{G', s{:}S, G''}{G', s{:}[S] \approx X{:}S, G''}$$

X fresh variable.

[$m1$] *membership*

$$\frac{G', X{:}S, G''}{(G', G'')\theta}$$

(i) $\theta = id$ if X variable, $getType(X) = S$ or
(ii) $\theta = id$ if X term, $leastSort(M, X) \leq S$ or
(iii) $\theta = \{X \mapsto Z{:}S'\}$ if Z fresh variable and $S' \in glbSorts(M, S, getType(X))$.

[$m2$] *membership*

$$\frac{G', f(\bar{s}){:}S, G''}{(G', (c',)(c,)\ G'')\theta}$$

where $(c)mb\ g(\bar{t}){:}T$ $(if\ c)$ is a fresh variant, with $T \leq S$, of a (conditional) membership in E, and θ A-unifier of $f'(\bar{s})$ and $g'(\bar{t})$ kinded variable terms, c' membership checks. i may be 0.

glbSorts is used in many rules, because when we try to unify one term with sort S and another term with sort T, the sort of the resulting unified term must be a common subsort of S and T, and $glbSorts(M, S, T)$ is the set of the supremes for this subsorts. This is a nondeterministic step, we can even find different answers depending on the sort that we choose, so we have to consider all possible sorts in $glbSorts(M, S, T)$. This issue has already been discussed by Hendrix and Meseguer in [HM12].

4.2 Examples

In the following examples we use the symbol $\rightsquigarrow_{[r]i}$ when we apply a calculus rule $[r]$. i is optional and may include the equation or membership applied as well as the A-unifier applied. We keep old variables in substitutions, when possible, to ease the reading of derivations. In real use, each substitution creates new variables on its right side to ensure idempotency. If no substitution is shown in a rule that needs it, id is assumed. We underline the subgoals where rules get applied.

Example 1

This example shows the necessity of membership checking in inference rules. We define natural numbers and multiples of number three:

```
fmod 3*NAT is
  sort Zero Nat .
  subsort Zero < Nat .
  op zero : -> Zero .
  op s_ : Nat -> Nat .
```

```
  sort 3*Nat .
  subsorts Zero < 3*Nat < Nat .
  var M3 : 3*Nat .
  mb (s s s M3) : 3*Nat .
endfm
```

If we try to solve the unification goal $s(Y{:}Nat){:}Nat = M3{:}3{*}Nat$, we get the following derivation:

1. $\underline{s(Y{:}Nat){:}Nat{=}M3{:}3{*}Nat} \leadsto_{[u]}$
2. $s(Y{:}Nat){:}3{*}Nat{\approx}Z{:}3{*}Nat, \underline{M3{:}3{*}Nat{\approx}Z{:}3{*}Nat} \leadsto_{[r]\ \theta=\{\underline{M3\mapsto Z:3*Nat}\}}$
3. $s(Y{:}Nat){:}3{*}Nat{\approx}Z{:}3{*}Nat, Z{:}3{*}Nat){:}3{*}Nat, \underline{(Z{:}3{*}Nat){:}3{*}Nat} \leadsto_{[m1]}$
4. $s(Y{:}Nat){:}3{*}Nat{\approx}Z{:}3{*}Nat, \underline{(Z{:}3{*}Nat){:}3{*}Nat} \leadsto_{[m1]}$
5. $\underline{s(Y{:}Nat){:}3{*}Nat{\approx}Z{:}3{*}Nat} \leadsto_{[r]\ \theta=\{\underline{Z\mapsto s(Y:Nat)}\}}$
6. $\underline{s(Y{:}Nat){:}3{*}Nat}, s(Y{:}Nat){:}3{*}Nat \leadsto_{[m2]\ mb\ sss(Z':3*Nat):3*Nat,\theta=\{\underline{Y\mapsto ssZ':3*Nat}\}}$
7. $\underline{ss(Z'{:}3{*}Nat){:}Nat} \leadsto_{[m1],\ leastSort(M,ss(Z':3*Nat))=Nat}$
8. $\square$

In step number five, we get the A-unifier θ because we ask for it at the kind level, dropping sorts of variables Y and Z. If we use the original sorts, *unify* returns `no unify` as answer, and we are not able to solve the unification problem. Also, if the membership condition had not been included in rule $[r]$, the derivation would have ended after step number five with answer $M3 \mapsto s(Y{:}Nat)$. With the membership condition we get the rest of the answer, $Y{:}Nat \mapsto ss(Z'{:}3{*}Nat)$.

Example 2

Let's see how conditional equations work. Consider the functional module:

```
fmod NAT-FIB is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op _<=_ : Nat Nat -> Bool .
  op f : Nat -> Nat .
  vars M N : Nat .
```

```
  eq  [a1] : 0 + N = N .
  eq  [a2] : s(M) + N = s(M + N) .
  eq  [e1] : 0 <= N = true .
  eq  [e2] : s(M) <= 0 = false .
  eq  [e3] : s(M) <= s(N) = M <= N .
  ceq [f1] : f(N) = s(0) if (N <= s(0)) .
  eq  [f2] : f(s(s(N))) = f(N) + f(s(N)) .
endfm
```

This is a version of Fibonacci's sequence, not the original one. Now we try to answer the goal $f(Y){:}Nat = s(s(0)){:}Nat$. Sorts and checks for memberships are omitted, since there is only one. The derivation is as follows:

1. $\underline{f(Y) = s(s(0))} \rightsquigarrow_{[u]}$
2. $\underline{f(Y) \approx X}, s(s(0)) \approx X \rightsquigarrow_{[t]}$
3. $\underline{f(Y) \approx Z}, Z \approx X, s(s(0)) \approx X \rightsquigarrow_{[r],\theta=\{X \mapsto s(s(0))\}}$
4. $\underline{f(Y) \approx Z}, Z \approx s(s(0)) \rightsquigarrow_{[n],[f2],\theta=\{\underline{Y \mapsto s(s(Y_1))}\}}$
5. $\underline{f(Y_1) + f(s(Y_1)) \approx Z}, Z \approx s(s(0)) \rightsquigarrow_{[i],\theta=\{Z \mapsto Z_1+Z_2\}}$
6. $\underline{f(Y_1) \approx Z_1}, f(s(Y_1)) \approx Z_2, Z_1 + Z_2 \approx s(s(0)) \rightsquigarrow_{[n],[f1],\theta=\{N \mapsto Y_1\}}$ **conditional!**
7. $\underline{Y_1 \leq s(0) \approx \mathit{true}}, s(0) \approx Z_1, f(s(Y_1)) \approx Z_2, Z_1 + Z_2 \approx s(s(0)) \rightsquigarrow_{[n],[e1],\theta=\{N \mapsto 0, \underline{Y_1 \mapsto 0}\}}$
8. $\underline{\mathit{true} \approx \mathit{true}}, s(0) \approx Z_1, f(s(0)) \approx Z_2, Z_1 + Z_2 \approx s(s(0)) \rightsquigarrow_{[r]}$
9. $\underline{s(0) \approx Z_1}, f(s(0)) \approx Z_2, Z_1 + Z_2 \approx s(s(0)) \rightsquigarrow_{[r],\theta=\{Z_1 \mapsto s(0)\}}$
10. $\underline{f(s(0)) \approx Z_2}, s(0) + Z_2 \approx s(s(0)) \rightsquigarrow_{[n],[f1],\theta=\{N \mapsto s(0)\}}$
11. $\underline{s(0) \leq s(0) \approx \mathit{true}}, s(0) \approx Z_2, s(0) + Z_2 \approx s(s(0)) \rightsquigarrow_{[n],[e3],\theta=\{M \mapsto 0, N \mapsto 0\}}$
12. $\underline{0 \leq 0 \approx \mathit{true}}, s(0) \approx Z_2, s(0) + Z_2 \approx s(s(0)) \rightsquigarrow_{[n],[e1],\theta=\{N \mapsto 0\}}$
13. $\underline{\mathit{true} \approx \mathit{true}}, s(0) \approx Z_2, s(0) + Z_2 \approx s(s(0)) \rightsquigarrow_{[r]}$
14. $\underline{s(0) \approx Z_2}, s(0) + Z_2 \approx s(s(0)) \rightsquigarrow_{[r],\theta=\{Z_2 \mapsto s(0)\}}$
15. $\underline{s(0) + s(0) \approx s(s(0))} \rightsquigarrow_{[n],[a2],\theta=\{M \mapsto 0, N \mapsto s(0)\}}$
16. $\underline{s(0 + s(0)) \approx s(s(0))} \rightsquigarrow_{[d]}$
17. $\underline{0 + s(0) \approx s(0)} \rightsquigarrow_{[n],[a1],\theta=\{N \mapsto s(0)\}}$

18. $\underline{s(0) \approx s(0)} \rightsquigarrow_{[r]}$
19. $\square$

From the underlined substitutions we get the desired answer $\sigma = \{Y \mapsto s(s(0))\}$.

Example 3

We consider now a specification of integer numbers with sum, difference, unary minus and a boolean comparison operation between integers `<=`. This is the functional module:

```
fmod INTEGERS is
  sort Int .
  op 0 : -> Int [ctor] .
  op s : Int -> Int [ctor] .
  op p : Int -> Int [ctor] .
  op _+_ : Int Int -> Int .
  op _-_ : Int Int -> Int .
  op -_ : Int -> Int .
  op _<=_ : Int Int -> Bool .
  vars N M : Int .
  eq [sp] : s(p(N)) = N .
  eq [ps] : p(s(N)) = N .
  eq [s1] : 0 + N = N .
  eq [s2] : s(M) + N = s(M + N) .
  eq [s3] : p(M) + N = p(M + N) .
  eq [d1] : N - 0 = N .
  eq [d2] : M - s(N) = p(M - N) .
  eq [d3] : M - p(N) = s(M - N) .
  eq [d4] : - N = 0 - N .
  eq [i1] : s(M) <= N = M <= p(N) .
  eq [i2] : p(M) <= N = M <= s(N) .
  eq [i3] : N <= N = true .
  eq [i4] : N <= p(N) = false .
  ceq [i5] : N <= s(M) = true if N <= M .
  ceq [i6] : N <= p(M) = false if N <= M == false .
endfm
```

Our goal now is $s(0) - X \leq s(0){:}Bool = true{:}Bool$. We will consider several derivations. We omit checking sorts again, since there is only one sort per kind:

1. $\underline{s(0) - X \leq s(0) = true} \rightsquigarrow_{[u]}$

2. $\underline{s(0) - X \leq s(0) \approx V}, true \approx V \rightsquigarrow_{[t]}$
3. $\underline{s(0) - X \leq s(0) \approx W}, W \approx V, true \approx V \rightsquigarrow_{[r]\ \theta=\{V \mapsto true\}}$
4. $\underline{s(0) - X \leq s(0) \approx W}, W \approx true \rightsquigarrow_{[i]\ \theta=\{W \mapsto Y \leq Z\}}$
5. $\underline{s(0) - X \approx Y}, s(0) \approx Z, Y \leq Z \approx true \rightsquigarrow_{[n],[d1],\ \theta=\{N \mapsto s(0), \underline{X \mapsto 0}\}}$
6. $\underline{s(0) \approx Y}, s(0) \approx Z, Y \leq Z \approx true \rightsquigarrow_{[r]\ \theta=\{Y \mapsto s(0)\}}$
7. $\underline{s(0) \approx Z}, s(0) \leq Z \approx true \rightsquigarrow_{[r]\ \theta=\{Z \mapsto s(0)\}}$
8. $\underline{s(0) \leq s(0) \approx true} \rightsquigarrow_{[n],[i3],\ \theta=\{N \mapsto s(0)\}}$
9. $\underline{true \approx true} \rightsquigarrow_{[r]}$
10. $\square$

The answer computed here is $\sigma = \{X \mapsto 0\}$.

Another derivation:

1. $\underline{s(0) - X \leq s(0) = true} \rightsquigarrow_{[u]}$
2. $\underline{s(0) - X \leq s(0) \approx V}, true \approx V \rightsquigarrow_{[n],[i5],\ \theta=\{N \mapsto s(0), M \mapsto X\}}$ **$\underline{\text{conditional!}}$**
3. $s(0) - X \leq 0 \approx true, true \approx V, \underline{true \approx V} \rightsquigarrow_{[r],\ \theta=\{V \mapsto true, M \mapsto X\}}$
4. $s(0) - X \leq 0 \approx true, \underline{true \approx true} \rightsquigarrow_{[r]}$
5. $\underline{s(0) - X \leq 0 \approx true} \rightsquigarrow_{[t]}$
6. $\underline{s(0) - X \leq 0 \approx W}, W \approx true \rightsquigarrow_{[i],\ \theta=\{W \mapsto W_1 \leq W_2\}}$
7. $s(0) - X \approx W_1, \underline{0 \approx W_2}, W_1 \leq W_2 \approx true \rightsquigarrow_{[r],\ \theta=\{W_2 \mapsto 0\}}$
8. $\underline{s(0) - X \approx W_1}, W_1 \leq 0 \approx true \rightsquigarrow_{[n],[d2],\ \theta=\{M \mapsto s(0), N \mapsto X', \underline{X \mapsto s(X')}\}}$
9. $\underline{p(s(0) - X') \approx W_1}, W_1 \leq 0 \approx true \rightsquigarrow_{[i]\ \theta=\{W_1 \mapsto p(W_1')\}}$
10. $\underline{s(0) - X' \approx W_1'}, p(W_1') \leq 0 \approx true \rightsquigarrow_{[n],[i5],\ \theta=\{N \mapsto s(0), \underline{X' \mapsto 0}\}}$
11. $\underline{s(0) \approx W_1'}, p(W_1') \leq 0 \approx true \rightsquigarrow_{[r],\ \theta=\{W_1' \mapsto s(0)\}}$
12. $\underline{p(s(0)) \leq 0 \approx true} \rightsquigarrow_{[n],[ps],\ \theta=\{N \mapsto 0\}}$
13. $\underline{0 \leq 0 \approx true} \rightsquigarrow_{[n],[i3],\ \theta=\{N \mapsto s(0)\}}$

14. $\underline{true \approx true} \rightsquigarrow_{[r]}$
15. $\square$

The answer computed here is $\sigma = \{X \mapsto s(0)\}$.

One incomplete derivation:

1. $\underline{s(0) - X \leq s(0) = true} \rightsquigarrow_{[u]}$
2. $\underline{s(0) - X \leq s(0) \approx V}, true \approx V \rightsquigarrow_{[t]}$
3. $s(0) - X \leq s(0) \approx W, W \approx V, \underline{true \approx V} \rightsquigarrow_{[r],\ \theta=\{V \mapsto true\}}$
4. $\underline{s(0) - X \leq s(0) \approx W}, W \approx true \rightsquigarrow_{[i],\ \theta=\{W \mapsto Y \leq Z\}}$
5. $s(0) - X \approx Y, \underline{s(0) \approx Z}, Y \leq Z \approx true \rightsquigarrow_{[r],\ \theta=\{Z \mapsto s(0)\}}$
6. $\underline{s(0) - X \approx Y}, Y \leq s(0) \approx true \rightsquigarrow_{[n],[d3],\ \theta=\{M \mapsto s(0), N \mapsto N', \underline{X \mapsto p(N')}\}}$
7. $\underline{s(s(0) - p(N')) \approx Y}, Y \leq s(0) \approx true \rightsquigarrow_{[i],\ \theta=\{Y \mapsto s(Y')\}}$
8. $\underline{s(0) - p(N') \approx Y'}, s(Y') \leq s(0) \approx true \rightsquigarrow_{[n],[d3],\ \theta=\{M \mapsto s(0), N \mapsto N'\}}$
9. $\underline{s(s(0) - N') \approx Y'}, s(Y') \leq s(0) \approx true \rightsquigarrow_{[i],\ \theta=\{Y' \mapsto s(Y'')\}}$
10. $\underline{s(0) - N' \approx Y''}, s(s(Y')) \leq s(0) \approx true \rightsquigarrow_{[n],[d1],\ \theta=\{N \mapsto s(0), \underline{N' \mapsto 0}\}}$
11. $\underline{s(0) \approx Y''}, s(s(Y'')) \leq s(0) \approx true \rightsquigarrow_{[r]\ \theta=\{Y'' \mapsto 0\}}$
12. $\underline{s(s(0)) \leq s(0) \approx true} \rightsquigarrow_{[n],[i1],\ \theta=\{M \mapsto s(0), N \mapsto s(0)\}}$
13. $\underline{s(0) \leq p(s(0)) \approx true} \rightsquigarrow_{[n],[i4],\ \theta=\{N \mapsto s(0)\}}$
14. $false \approx true$

No further derivation is possible. We couldn't prove that $\sigma = \{X \mapsto p(0)\}$ is an answer, this way (most of all because it is not an answer). In this example narrowing will never stop, it can try every negative integer without proving that it is an answer, because narrowing cannot perform inductive proving.

Chapter 5

Correctness of the calculus for unification

We prove correctness of the calculus with respect to normalized idempotent substitutions for the executable MEL theory $(\Sigma, E \cup A)$ and the corresponding rewrite theory $\mathcal{R}_E = (\Sigma', A, R_E)$ associated to it (both are equivalent).

5.1 Soundness

We prove that given a unification goal G, if $G \rightsquigarrow_\sigma^* \square$ then $G\sigma$ can be derived, so σ is a solution for G. Soundness of the calculus is proved by induction on the length of the derivation. Recall that all calculus rules always check the correct typing. We transform any goal $(s{:}S \ op \ t{:}T)$ into $(s \ op \ t, \ s{:}S, \ t{:}T)$, as explained in chapter 4.

Base step: proofs with length one. We have a goal with one element. The only inference rules that delete goals without creating new ones are $[m1]$, and $[m2]$ in the case of constants $(i = 0)$ and non conditional memberships:

$[m1]$ *membership*

$$\frac{X{:}S}{\square}$$

(i) $\theta = id$ if X variable, $getType(X) = S$ or
(ii) $\theta = id$ if X term, $leastSort(M, X) \leq S$ or
(iii) $\theta = \{X \mapsto Z{:}S'\}$ if Z fresh variable and $S' \in glbSorts(M, S, getType(X))$.

If X is a term and $leastSort(M, X) \leq S$ we can derive $X{:}S$. If X is a variable then if $S'{=}S$, $X{:}S \in \Sigma$. Otherwise if X has sort T, θ is a valid substitution because

$S' \leq T$. If S' is a subsort of S, $Z{:}S' \in \Sigma$, $(\forall y{:}[S])y{:}S$ *if* $y{:}S') \in \mathcal{E}$, so we can derive $Z{:}S$. If $S'{=}S$ then $Z{:}S \in \Sigma$.

□

[$m2$] *membership,* $i = 0$ *no conditions*

$$\frac{c{:}S}{\square}$$

where *mb* $c{:}T$ is a membership (explicit or implicit) in E, with $T \leq S$.

As $T \leq S$ we have added the membership *cmb* $X{:}S$ *if* $X{:}T$. As $c{:}T$, by membership we derive $c{:}S$.

□

Induction step: We assume that if a derivation of □ from a goal G, with length n or less, provides a substitution σ, then $G\sigma$ is derivable and the associated reachability goal rewrites to $\top$, that is, σ is an answer of G. We have to prove that this property holds for derivations with length $n + 1$. We assume $G \equiv g, G'$ (G' may be empty), and check all possible calculus rules applied to g:

[u] *unification*

$$\frac{s{:}S{=}t{:}T, G'}{s{:}S'{\approx}X{:}S', t{:}S'{\approx}X{:}S', G'}$$

where X fresh variable, $S' \in \mathit{glbSorts}(M, S, T)$.

By induction hypothesis if there is a normalized idempotent substitution σ, computed answer for $s{:}S' \approx X{:}S'$, $t{:}T \approx X{:}S'$ and G' we can derive $s\sigma{\to}X\sigma$, $t\sigma{\to}X\sigma$, $s\sigma{:}S'$ $t\sigma{:}S'$ and $G'\sigma$. Since $S' \leq S$ and $S' \leq T$, we can derive $s\sigma{:}S$ and $t\sigma{:}T$.
$g\sigma \equiv s\sigma{:}S{=}t\sigma{:}T$. We are going to show that we can solve the equivalent problem in $\mathcal{R}_E$: $\mathit{eq}(s\sigma{:}S, t\sigma{:}T) \to \top$. As $s\sigma{:}S$ and $t\sigma{:}T$ are derivable all that is left to do is proving that $\mathit{eq}(s\sigma, t\sigma) \to \top$. From $\mathit{eq}(X, X) \to \mathit{tt}$ by replacement with $X \mapsto X\sigma$ we get $\mathit{eq}(X\sigma, X\sigma) \to \mathit{tt}$. Applying congruence twice we get: $\mathit{eq}(s\sigma, t\sigma) \to \mathit{eq}(X\sigma, X\sigma)$. Then, by transitivity, $\mathit{eq}(s\sigma, t\sigma) \to \top$.
σ is a solution of g and also of G', so σ is a solution of G.

□

[x] *matching*

$$\frac{s{:}S := t{:}T, G'}{t{:}S'{\approx}s{:}S', G'}$$

where $S' \in \mathit{glbSorts}(M, S, T)$.

By I.H. if σ is a computed answer of $t{:}S' \approx s{:}S'$ and G', we derive $s\sigma{:}S$ and $t\sigma{:}T$, as before, and $t\sigma \to s\sigma$. Recall that in a MEL theory we treat matching logically as equality, the difference between them is computational, so $g\sigma \equiv s\sigma{:}S = t\sigma{:}T$. Again, we show that we can solve the equivalent problem in $\mathcal{R}_E$: $eq(s\sigma{:}S, t\sigma{:}T) \to \top$. As we can derive $s\sigma{:}S$ and $t\sigma{:}T$, we have to prove $eq(s\sigma, t\sigma) \to \top$. From $eq(X, X) \to tt$ by replacement with $X \mapsto s\sigma$ we have $eq(s\sigma, s\sigma) \to \top$. By congruence, $eq(s\sigma, t\sigma) \to eq(s\sigma, s\sigma)$. Then, by transitivity, $eq(s\sigma, t\sigma) \to \top$.
σ is a solution of $s{:}S := t{:}T$ and also of G', so σ is a solution of G.

□

[n] *narrowing*

$$\frac{s{:}S \approx t{:}T, G'}{((c',)s{:}S', (c,), r{:}S' \approx t{:}S', G')\theta}$$

(c)eq $l = r$ $(if\ c) \in E$ has fresh variables, $S' \in glbSorts(M, S, T)$,
θ A-unifier of s' and l' kinded variable terms, c' membership checks.

$\sigma \equiv \theta\sigma'$ is a computed answer for c', $s{:}S'$, c, $r{:}S' \approx t{:}S'$ and G'. By I.H. we can derive $c'\sigma$, meaning that for every instantiated variable in θ we can derive that the value it has been assigned to in σ is of the right sort. As before, we derive $s\sigma{:}S$, $r\sigma{:}R$ and $t\sigma{:}T$. $s\theta =_A l\theta$ implies $s\sigma =_A l\sigma$, with every instantiated variable in θ of the right sort. By reflexivity, $s\sigma \to_A l\sigma$. $c\sigma$ is derivable so we can derive $l\sigma \to r\sigma$, by replacement. By transitivity we derive $s\sigma \to r\sigma$. By I.H. $r\sigma \to t\sigma$ is also derivable. Then, again by transitivity, $s\sigma \to t\sigma$. Is is important to remember that M is ACU-coherence completed, so even if the original left term l cannot be A-unified with s, the ACU-coherence completed version of l may A-unify with s, because it allows checking any possible reordering of the subterms of s.

□

[t] *transitivity*

$$\frac{s{:}S \approx t{:}T, G'}{s{:}S' \approx X{:}S', X{:}S' \approx t{:}S', G'}$$

where X fresh variable, $S' \in glbSorts(M, S, T)$.

If σ is the computed answer, by I.H. we can derive $s \to X$ and $X \to t$ with correct typing as before. Then, by transitivity, we can derive $s \to t$.

□

[i] *imitation*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \approx X{:}T, G'}{s_i{:}S_i \approx X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S', f(\bar{X}{:}\bar{S}){:}S', G'\theta}$$

$$\text{where } X \notin Var(s), \theta = \{X \mapsto f(\bar{X}{:}\bar{S})\},$$
$$X_i \text{ fresh variables, } S' \in glbSorts(M, S, T).$$

$\sigma \equiv \theta\sigma'$, σ' computed answer for $s_i{:}S_i \approx X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S, f(\bar{X}{:}\bar{S}){:}T$ and $G'\theta$ as before. By I.H. we can derive $X_i\sigma'{:}S_i$ and $s_i\sigma'{:}S_i$.
As $f(s\sigma) \equiv f(s\theta\sigma') \equiv f(s\sigma')$ $(X \notin Var(s))$, by I.H. we can derive $f(\bar{s}\sigma{:}\bar{S}){:}S$.
$X\sigma \equiv X\theta\sigma' \equiv f(\bar{X}\sigma')$ so, by I.H., we can derive $X\sigma{:}T$.
By I.H. $s_i\sigma' \to X_i\sigma'$. Then, by congruence, $f(\bar{s}\sigma') \to^* f(\bar{X}\sigma') \equiv X\sigma$.
So we can derive $g\sigma$, and σ' answer of $G'\theta$ implies can derive $G'\theta\sigma'$, that is, we can also derive $G'\sigma$.

□

[d] *decomposition*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \approx f(\bar{t}{:}\bar{T}){:}T, G'}{s_1{:}S_1 \approx t_1{:}T_1, ..., s_n{:}S_n \approx t_n{:}T_n, f(\bar{s}{:}\bar{S}){:}S', f(\bar{t}{:}\bar{T}){:}S', G'}$$

$$\text{where } S' \in glbSorts(M, S, T).$$

If σ is a computed solution, then by I.H. we derive $f(\bar{s}\sigma{:}\bar{S}){:}S$, $f(\bar{t}\sigma{:}\bar{T}){:}T$, $s_i\sigma \to t_i\sigma$, $s'_i\sigma{:}S'_i$, $t'_i\sigma{:}T'_i$, as before. By congruence we derive $f(\bar{s}\sigma) \to f(\bar{t}\sigma)$.

□

[r] *removal of equations*

$$\frac{s{:}S \approx t{:}T, G'}{((c',)s{:}S', t{:}S', G')\theta}$$

$$\theta \text{ } A\text{-unifier of } s' \text{ and } t' \text{ kinded variable terms,}$$
$$c' \text{ membership checks, } S' \in glbSorts(M, S, T).$$

$\theta\sigma' \equiv \sigma$ is a computed answer for G', c', $s{:}S'$ and $t{:}S'$. By I.H. σ is a solution for G', c', $s{:}S'$ and $t{:}S'$.

$s\sigma =_A t\sigma$ and $c'\sigma$ is derivable, so variable assignments have correct checking. By reflexivity we derive $s\sigma \to t\sigma$. Then σ is a solution of $s{:}S \to t{:}T$, so σ is a solution for G.

□

[s] *subject reduction*

$$\frac{t{:}S, G'}{t{:}[S] \approx X{:}S, G'}$$

$$X \text{ fresh variable.}$$

If σ is a computed answer then, by I.H., we can derive $t\sigma \rightarrow X\sigma$ and $X\sigma{:}S$. Applying subject reduction we get $t\sigma{:}S$.

□

[$m1$] *membership*

$$\frac{X{:}S, G'}{G'\theta}$$

(i) $\theta = id$ if X variable, $getType(X) = S$ or
(ii) $\theta = id$ if X term, $leastSort(M, X) \leq S$ or
(iii) $\theta = \{X \mapsto Z{:}S'\}$ if Z fresh variable and $S' \in glbSorts(M, S, getType(X))$.

By induction hypothesis if $\theta\sigma' \equiv \sigma$ is a computed answer for G', σ is a solution for G'.
As seen in the base case, $X\theta{:}S$ is derivable, so any instantiation of X in σ' with sort lower or equal than S or S', depending on the case, is also derivable.

□

[$m2$] *membership*

$$\frac{f(\bar{s}){:}S, G'}{((c',)(c,)\, G')\theta}$$

where $(c)mb\ g(\bar{t}){:}T$ $(if\ c)$ is a fresh variant, with $T \leq S$, of a (conditional) membership in E, and θ A-unifier of $f'(\bar{s})$ and $g'(\bar{t})$ kinded variable terms, c' membership checks. i may be 0.

$\theta\sigma' \equiv \sigma$ is a computed answer of c', c and G'. By I.H. σ is a solution for c', c and G'.
$f'(s\theta) =_A g'(t\theta)$ and $c'\sigma$ derivable implies $f(s\sigma) =_A g(t\sigma)$. Then by membership, as $c\sigma$ is derivable, we derive $f(\bar{s}\sigma){:}S$.

□

5.2 Completeness

As previously stated, we assume that we have an A-unification algorithm that returns order sorted unifiers, in Maude it is called *unify* and gives a CSU, so our substitutions may be more general than the ones we are imitating, therefore improving the answer. We also have the *kind* function, that returns the kind of a term.

We prove that if ρ is a normalized idempotent answer of G ($G\rho \rightarrow^* \top$), then there is ρ' normalized idempotent, with $\rho \ll_{\mathcal{E}} \rho'$, such that $G \rightsquigarrow_{\rho'} \Box$. Completeness of the calculus is proved by induction on the length of inferences in

$\mathcal{R} = (\Sigma', A, R_E)$, looking at the last inference rule used:
Base step:

(Reflexivity)

$$\frac{}{s \to t}$$
$$\text{if } s =_A t$$

$s\rho =_A t\rho$ allows the inference $s\rho \to t\rho$. Any instantiated variable x_i in ρ must have correct type S_i, that is, we have derived the correct type for it. By I.H we can compute some σ, with $\rho \ll_{\mathcal{E}} \sigma$, answer of $s' =_A t', \bigwedge x_i{:}S_i$, that is, if we drop sorts on inner variables of s and t we get a more general answer for s' and t'. σ being more general that ρ means that the instantiated variables in σ are a subset of those in ρ. Then all the instantiated variables in σ have correct type and $s\sigma{:}kind(M,s) =_A t\sigma{:}kind(M,t)$.
$s{:}kind(M,s) \approx t{:}kind(M,t) \rightsquigarrow_{[r],\sigma} s\sigma{:}kind(M,s), t\sigma{:}kind(M,t) \rightsquigarrow^* \square$.

On the rest of rules when no membership gets involved we omit the part on kinds.

□

Induction step:

(Transitivity)

$$\frac{t_1 \to t_2,\ t_2 \to t_3}{t_1 \to t_3}$$

$t_1\rho \to t_2$ and $t_2 \to t_3\rho$ allows the inference $t_1\rho \to t_3\rho$, with ρ idempotent. Then id is a solution for the reachability problems $t_1\rho \to t_2$ and $t_2 \to t_3\rho$. As the derivation of both terms is smaller than that of $t_1\rho \to t_3\rho$, by I.H. we can compute an idempotent answer lower or equal than id (so it must be id) for $t_1\rho \approx t_2$ and $t_2 \approx t_3\rho$, that is, we can compute $t_1\rho \approx t_2$ and $t_2 \approx t_3\rho$.
Then, by rule $[t]$, we can compute $t1\rho \approx t_3\rho$, and ρ is a computed solution for $t_1 \to t_3$.

□

(Congruence)

$$\frac{t_i \to t_i'}{f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, t_i', \ldots, t_n)}$$

From $t_i\rho \to t_i'\rho$ we derive $f(t_1, \ldots, t_i\rho, \ldots, t_n) \to f(t_1, \ldots, t_i'\rho, \ldots, t_n)$. By I.H. there is σ, with $\rho \ll_{\mathcal{E}} \sigma$, such that σ is a computed solution for $t_i\sigma \approx t_i'\sigma$. Without loss of generality we assume i=1.
$f(t_1, \ldots, t_n) \approx f(t_1', \ldots, t_n) \rightsquigarrow_{[d]} t_1 \approx t_1', \ldots, t_n \approx t_n \rightsquigarrow^*_{[r]} t_1 \approx t_1' \overset{I.H.}{\rightsquigarrow}_\sigma \square$.

□

(Replacement)

$$\frac{A_1^\bullet\sigma \ldots A_n^\bullet\sigma}{u \to t'\sigma}$$

$$\text{if } t \to t' \textit{ if } A_1 \ldots A_n \text{ in } \mathcal{R}_E \text{ and } u =_A t\sigma$$

If ρ is a solution then $u\rho =_A t\sigma$ and $A_1^\bullet\sigma \ldots A_n^\bullet\sigma$. By I.H., there is σ', with $\sigma \ll_{\mathcal{E}} \sigma'$ such that σ' is a computed solution for $A_1 \wedge \ldots \wedge A_n$. Then we derive $t\sigma' \approx t'\sigma'$. σ is an instantiation of variables from σ', so we can also derive $t\sigma \approx t'\sigma$. As $u\rho =_A t\sigma$, there is ρ', with $\rho \ll_{\mathcal{E}} \rho'$, A-match for u and $t\sigma$. By removal of equations, we derive $u\rho' \approx t\sigma$ and, by transitivity, we derive $u\rho' \approx t'\sigma$.

□

(Subject Reduction)

$$\frac{t \to t', t' : s}{t : s}$$

If ρ is a solution, by I.H. there is σ, with $\rho \ll_{\mathcal{E}} \sigma$, such that σ is a computed answer for $t \approx t'$ and $t' : s$. Then, by subject reduction, σ is a computed answer for $t{:}s$.

□

(Membership)

$$\frac{A_1^\bullet\sigma \ldots A_n^\bullet\sigma}{u : s}$$

$$\text{if } t : s \textit{ if } A_1 \ldots A_n \text{ in } \mathcal{R}_E \text{ and } u =_A t\sigma$$

If ρ is a solution then $u\rho =_A t\sigma$ and $A_1^\bullet\sigma \ldots A_n^\bullet\sigma$. By I.H., there is σ', with $\sigma \ll_{\mathcal{E}} \sigma'$, such that σ' is a computed solution for $A_1 \wedge \ldots \wedge A_n$. By membership σ' is a computed solution for $t{:}s$, that is $t\sigma'{:}s$. $t\sigma$ is an instantiation of variables from $t\sigma'$, so $t\sigma{:}s$ too. $u\rho =_A t\sigma$ implies we get ρ', with $\rho \ll_{\mathcal{E}} \rho'$, A-match for u and $t\sigma$. By subject reduction we get $u\rho'{:}s$, so ρ' is a computed solution for $u{:}s$.

□

Chapter 6

Transformations for unification

The calculus described in chapter 4 assumes that we have an oracle that always chooses the right rule, the right equation and also, if needed, the right A-unifier to apply at each step of the calculus, so we can get an answer for our unification problem. What we are going to see now is a set of transformations that can find the same answer computationally. While the calculus is concerned with how problems evolve, the transformations are concerned with how to make problems evolve, that is, the logic for a future implementation. The transformations show us what structures do we need, how they relate one to another, where do we keep intermediate results, etc. They are pretty much like algorithms: the details of the implementation are omitted, so we can clearly see the more important parts of the computation.

An E-unification problem from our calculus will be here a list of goals, each one a possible computation of the answer, separated by symbol $\|$, and a last element A, the list of found answers, separated by symbol $\triangle$. Each goal contains a list of subgoals, where each subgoal is surrounded by square brackets, and a partial solution σ computed so far. σ is placed after the list of subgoals, separated by a symbol ∇. Each subgoal holds inference rules and may hold equations or memberships that apply to it. Rules for problem transformation take one subgoal of a goal and may modify it, delete it, or add a new goal to the problem, based on this one. The new goal will have additional braces when applying conditional equations or memberships, or when applying some rules, delaying the development of some subgoals until others are solved. We turn the original problem into a reachability problem: is it possible to reach a configuration in which one of the goals has an empty list of subgoals? This goal would hold a substitution σ that would be one solution to our E-unification problem.

6.1 Transformation rules for unification

We keep the whole computation in one structure. Each subgoal on it has the form $G \,\square\, R \,\square\, E$ or $G \,\square\, R \,\square\, M$, where:

- G is the actual subgoal. It can be unification, matching, equation or membership.
- R is a queue/set of inference rules. Each element holds a rule of the calculus and a number for the A-unifier to request for that rule, if needed. If the list gets empty, the subgoal is exhausted, and we delete the goal.
- E is a queue/set of pairs. Each element holds an equation and the number of the A-unifier to request for that equation. This list is used by rule **NA**.
- M is a queue/set of memberships, conditional or unconditional. Each element holds a membership and the number of A-unifier to ask for. The list is used by rule **ME**.

As an example, the following structure:

$$[+(a,c){:}S \,\square\, s, m0, m1, m2 \,\square\, (a,0),(e,0)]$$

$$[+(d,+(b,e)){:}S \approx +(+(a,c),b){:}S \,\square\, d,n,t,(r,0) \,\square\, (e1,0)]$$

$$\nabla\, \{X \mapsto a\} \,\triangle\, \mathit{empty}$$

has one goal with two subgoals, $+(a,c){:}S$ and $+(d,+(b,e)){:}S \approx +(+(a,c),b){:}S$, each one with the calculus rules, equations and memberships that may apply to them. There is one partial computed answer $\{X \mapsto a\}$, but the set of computed answers remains *empty*.

One or several subgoal may have $\Diamond$ instead of $\square$ as separator, with braces surrounding them. This means that the subgoals are delayed, and depend on other subgoals, inside other braces too but with $\square$ as separator, to resume computation. That is, there are at least two pairs of braces, and delayed subgoals are always inside a nested pair of braces. This example:

$$\{[+(a,c){:}S \,\square\, s, m0, m1, m2 \,\square\, (a,0),(e,0)]$$

$$\{[+(d,+(b,e)){:}S \approx +(+(a,c),b){:}S \,\Diamond\, d,n,t,(r,0) \,\Diamond\, (e1,0)]\}\}$$

$$\nabla\, \{X \mapsto a\} \,\triangle\, \mathit{empty}$$

is similar to the previous example, but now we cannot apply any transformation to the subgoal $+(d,+(b,e)){:}S \approx +(+(a,c),b){:}S$ until we solve the subgoal $+(a,c){:}S$.

The initial goal contains several subgoals $s_i{:}S_i{=}t_i{:}T_i$, pairs of terms we want to unify. For each subgoal R holds u, E is empty, and σ holds the *id* substitution. The list A of found answers is empty.

These are the rules for problem transformation in this approach, where E, M stand for fresh variants of the queues/sets of equation pairs, and membership pairs, and R is the initial queue/set of inference rules, but each list restricted each time to the applicable inference rules, equations, or membership rules for the subgoal. We have omitted sorts on terms, when not strictly necessary, to improve readability. Rules can be applied inside any number of nested levels of braces, except **AF** that applies to the whole computation. $\bar{g}$ stands for the rest of subgoals of the goal we are considering, $\bar{g'}$ stands for some nested delayed subgoals, $\bar{G}$ are the rest of concurrent computations. All of them may be empty:

AF Answer Found

$$\nabla\ \sigma \parallel \bar{G} \triangle A \Vdash_{AF} \bar{G} \triangle \sigma, A$$

G may be empty.

RC Resume Computation

$$\{\{[g_i \Diamond R_i \Diamond \ldots]_{i=1}^{n}\{\bar{g'}\}\}\ \bar{g}\ \nabla\ \sigma \parallel \bar{G} \triangle A \Vdash_{RC}$$
$$\{[g_i \Box R_i \Box \ldots]_{i=1}^{n}\{\bar{g'}\}\ \bar{g}\ \nabla\ \sigma \parallel \bar{G} \triangle A$$

UN UNification

$$[s{:}S{=}t{:}T \Box u \Box]\ \bar{g}\ \nabla\ \sigma \parallel \bar{G} \triangle A \Vdash_{UN}$$
$$\bar{G}\ (\parallel [s{:}S'_j{\approx}X_j{:}S'_j \Box R_1 \Box E_1]\ [t{:}S'_j{\approx}X_j{:}S'_j \Box R_2 \Box E_2]\ \bar{g}\ \nabla\ \sigma)_{j=1}^{m} \triangle A$$

X_j fresh variables, $\{S'_j\}_{j=1}^{m}{=}\mathit{glbSorts}(M,S,T)$.

MA MAtching

$$[s{:}S := t{:}T \Box x \Box]\ \bar{g}\ \nabla\ \sigma \parallel \bar{G} \triangle A \Vdash_{MA}$$
$$\bar{G}\ (\parallel [t{:}S'_j \approx s{:}S'_j \Box R \Box E]\ \bar{g}\ \nabla\ \sigma)_{j=1}^{m} \triangle A$$

$$\text{where } \{S'_j\}_{j=1}^{m} = \mathit{glbSorts}(M,S,T).$$

NA NArrowing

$$[s{:}S \approx t{:}T \,\square\, n, \bar{r} \,\square\, (e_1, n), \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{NA}$$
$$\bar{G} \parallel [s{:}S \approx t{:}T \,\square\, \bar{r}, n \,\square\, \bar{e}, (e_1, n{+}1)]\ \bar{g} \,\nabla\, \sigma$$
$$(\parallel \{[c_1\theta \,\square \ldots](\{[c_i\theta \,\Diamond \ldots])_{i=2}^{p}\{[r\theta{:}S'_j \approx t\theta{:}S'_j \,\Diamond\, R \,\Diamond\, E]\} \ldots\}\ \bar{g}\theta \,\nabla\, \sigma\theta)_{j=1}^{m} \,\triangle\, A$$

where $e_1 \equiv$ (c)eq $l = r$ $(if\ c)$ is a fresh variant of a (conditional) equation in E, θ n^{th} A-unifier of s' and l', $c' \cup c{=}\{c_i\}_{i=1}^{p}$, $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$.

$$[s \approx t \,\square\, n, \bar{r} \,\square\, (e_1, n), \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{NA} [s \approx t \,\square\, n, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

if $e_1 \equiv$ (c)eq $l = r$ $(if\ c)$, and no n^{th} A-unifier of s' and l' exists.

$$[s \approx t \,\square\, n, \bar{r} \,\square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{NA} [s \approx t \,\square\, \bar{r} \,\square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

TR TRansitivity

$$[s{:}S \approx t{:}T \,\square\, t, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{TR}$$
$$\bar{G} \parallel [s{:}S \approx t{:}T \,\square\, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma$$
$$(\parallel \{[s{:}S'_j \approx X_j{:}S'_j \,\square\, R \,\square\, E]\{[X_j{:}S'_j \approx t{:}S'_j \,\Diamond\, R \,\Diamond\, E]\}\}\ \bar{g} \,\nabla\, \sigma)_{j=1}^{m} \,\triangle\, A$$

where $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$, X_j fresh variables.

IM IMitation

$$[f(\bar{s}{:}\bar{S}){:}S \approx X{:}T \,\square\, i, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{IM}$$
$$\bar{G} \parallel [f(\bar{s}{:}\bar{S}){:}S \approx X{:}T \,\square\, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma$$
$$(\parallel \{[s_k{:}S_k \approx X_k{:}S_k \,\square\, R \,\square\, E]_{k=1}^{l}\{[f(\bar{s}{:}\bar{S}){:}S'_j \,\Diamond\, R \,\Diamond\, M]\}\}\ \bar{g}\theta \,\nabla\, \sigma\theta)_{j=1}^{m} \,\triangle\, A$$

where $X \notin Var(s)$, $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T, getType(X))$, $\theta = \{X \mapsto f(\bar{X})\}$, X_k fresh variables $(k = 1 \ldots l)$.

EL ELimination

$$[s{:}S \approx t{:}T \,\square\, (r, n), \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EL}$$
$$\bar{G} \parallel [s{:}S \approx t{:}T \,\square\, \bar{r}, (r, n{+}1) \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma$$
$$(\parallel [X_i\theta{:}S_i \,\square\, R \,\square\, M]_{i=1}^{n}\ [s\theta{:}S'_j \,\square\, R \,\square\, M]\ [t\theta{:}S'_j \,\square\, R \,\square\, M]\ \bar{g}\theta \,\nabla\, \sigma\theta)_{j=1}^{m} \,\triangle\, A$$

where $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$, and $\theta = n^{th} A$-unifier of s' and t', $c'{=}\{X_i{:}S_i\}_{i=1}^{n}$.

$$[s \simeq t \,\square\, (r, n), \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EL} [s \simeq t \,\square\, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

if no $\theta = n^{th} A$-unifier of s' and t' exists.

$$[s \simeq t \,\square\, \square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EL} \bar{G} \,\triangle\, A$$

$$\{\}\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EL} \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

DC DeComposition

$$[f(\bar{s}{:}\bar{S}){:}S \approx f(\bar{t}{:}\bar{T}){:}T \,\square\, d, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{DC}$$
$$\bar{G} \parallel [f(\bar{s}){:}S \approx f(\bar{t}){:}T \,\square\, \bar{r} \,\square\, \bar{e}]\ \bar{g} \,\nabla\, \sigma$$
$$(\ \parallel \{[s_i{:}S_i \approx t_i{:}T_i \,\square\, R_i \,\square\, E_i]_{i=1}^{n}\{[f(\bar{s}){:}S'_j \,\Diamond\, R \,\Diamond\, M]\}\ \bar{g}\} \,\nabla\, \sigma)_{j=1}^{m} \,\triangle\, A$$

where $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$.

SR Subject Reduction

$$[s{:}S \,\square\, s, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{SR}$$
$$\bar{G} \parallel [s{:}S \,\square\, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel [s{:}[S] \approx X{:}S \,\square\, R \,\square\, E]\ \bar{g} \,\nabla\, \sigma \,\triangle\, A$$

X fresh variable.

ME MEmbership

$$[s{:}S \,\square\, m0, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME} \bar{G} \,\triangle\, A$$

If s ground term with sort S' when reduced, and $S' \not\leq S$.

$$[s{:}S \,\square\, m0, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME} \bar{G} \parallel \bar{g} \,\nabla\, \sigma \,\triangle\, A$$

If s ground term with sort S' when reduced, and $S' \leq S$.

$$[X{:}S \,\square\, m1, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME} \bar{G}\ (\parallel \bar{g}\theta_j \,\nabla\, \sigma\theta_j)_{j=1}^{m} \,\triangle\, A$$

(i) $\theta_1 = id$ if X variable, $getType(X) \leq S$, or
(ii) $\theta_1 = id$ if X term, $leastSort(M, X) \leq S$, or
(iii) $\theta_j = \{X \mapsto Z_j{:}S'_j\}$, Z_j fresh variables and $\{S'_j\}_{j=1}^{m}{=}glbSorts(M, S, getType(X))$.

$$[X{:}S \,\square\, m1, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME} [X{:}S \,\square\, \bar{r} \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

If conditions above do not hold.

$$[f(\bar{s}){:}S \,\square\, m2, \bar{r} \,\square\, (m_2, n), \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME}$$
$$\bar{G} \parallel [f(\bar{s}){:}S \,\square\, \bar{r}, m2 \,\square\, \bar{m}, (m_2, n{+}1)]\ \bar{g} \,\nabla\, \sigma \parallel$$
$$\parallel \{[c_1\theta \,\square\, \ldots](\{[c_i\theta \,\Diamond\, \ldots])_{i=2}^{n}\}\ldots\}\ \bar{g}\theta \,\nabla\, \sigma\theta \,\triangle\, A$$

where $m_2 \equiv (c)mb\ g(\bar{t}){:}T$ $(if\ c)$ is a fresh variant, with $T \leq S$, of a (conditional) membership in E, and θ n-th A-unifier of $f'(\bar{s})$ and $g'(\bar{t})$, $c' \cup c{=}\{c_i\}_{i=1}^{n}$. i may be 0.

$$[f(\bar{s}{:}\bar{S}){:}S \,\square\, m2, \bar{r} \,\square\, (m_2, n), \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{ME}$$
$$[f(\bar{s}{:}\bar{S}){:}S \,\square\, \bar{r}, m2 \,\square\, \bar{m}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

if no n-th A-unifier of $f'(\bar{s})$ and $g'(\bar{t})$ exists.

$$[f(\bar{s}{:}\bar{S}){:}S \,\square\, m2, \bar{r}\,\square\,]\ \bar{g} \nabla \sigma \parallel \bar{G} \triangle A \Vdash_{ME}$$
$$[f(\bar{s}{:}\bar{S}){:}S \,\square\, \bar{r}\,\square\,]\ \bar{g} \nabla \sigma \parallel \bar{G} \triangle A$$

$$[f(\bar{s}{:}\bar{S}){:}S \,\square\,\square\,]\ \bar{g} \nabla \sigma \parallel \bar{G} \triangle A \Vdash_{ME} \bar{G} \triangle A$$

We explain in detail some of the rules:

- Rule **UN** turns one goal with a subgoal $s{:}S{=}t{:}T$ into several goals each one having this subgoal substituted by a different pair of subgoals $s{:}S'_j{\approx}X_j{:}S'_j$, $t{:}S'_j{\approx}X_j{:}S'_j$. Each sort S'_j is an element of $glbSorts(M, S, T)$, a different *greatest lower bound* of S and T with respect to M.

- Rule **NA** takes one goal with a subgoal $s{:}S \approx t{:}T$, increases index n of the applied equation $l = r$ *if* c, if there exists th n-th A-unification θ of s and l, and creates several goals each one having different sort as in the rule above. On each goal the subgoal is replaced by checks for sorts of instantiated inner variables, checks for conditions of the applied equation, a check for right typing of the whole term $s\theta{:}S'$ and a new unification problem $r\theta{:}S' \approx t\theta{:}S'$. If there is no n-th A-unification, the equation is erased from the subgoal, and if there are no equations left to apply, the calculus rule n is erased from the subgoal.

- The first case of rule **ME** $(m0)$ is special: it is not present in the original calculus rules. This is because this is a deletion rule and the calculus rules are meant to develop the calculus, we always have an oracle, not to drop some part of it, but the transformation rules try to reflect the kwnowledge that we have about the problem at any moment, and this rule reflects the case when we know that we have reached a dead end. It represents one case in which we can be sure that the corresponding goal can not be achieved. The subgoal is a term s that must have sort S, but it is ground, so there are no variables that we can instantiate, and when reduced its sort is some S' which is not a subsort of S (S included). Then this subgoal is a failure, and we drop the whole goal. The rest of the cases where we drop goals is by exhaustion of rules to apply, but this rule, being not strictly necessary, helps cutting down the size of the computations by early identification of fail subgoals. The second case of rule **ME** $(m0)$ is also covered by rule **ME** $(m1)$, but we write it down here because in this way the implementation will only need rule **ME** $(m0)$ in the case of ground terms.

6.2 Example

In this ACU-coherence completed functional module:

```
fmod AC is
  sort S .
  ops a b c d e : -> S .
  var Y : S .
  op + : S S -> S [comm assoc] .
  eq +(a, c) = +(d, e) [label e1] .
  eq +(a, c, Y) = +(d, e, Y) [label e1C] .
endfm
```

we try to solve he E-unification problem $+(X{:}S, +(b, c)){:}S = +(d, +(b, e)){:}S$.

We have implicit memberships $mb\ a{:}S, \ldots, mb\ e{:}S$, and $mb\ +(X{:}S, Y{:}S){:}S$. We call them $a, \ldots, e, +$. This is part of the computation, where each G_i is the list of goals not shown from previous steps (sorts are again omitted when not necessary). Chosen rules, equations and memberships are underlined. If no substitution is shown in a rule that needs it, *id* is assumed:

1. $[+(X, +(b, c)){:}S = +(d, +(b, e)){:}S \,\square\, \underline{u} \,\square] \,\nabla\, id \,\triangle\, empty \Vdash_{UN}$

2. $[+(X, +(b, c)){:}S \approx Z{:}S \,\square\, \underline{(i, 0)}, n, t, (r, 0) \,\square\, (e1, 0)]$
   $[+(d, +(b, e)){:}S \approx Z{:}S \,\square\, \overline{(i, 0)}, n, t, (r, 0) \,\square\, (e1, 0)]$
   $\nabla\, id \parallel G_1 \,\triangle\, empty \Vdash_{NA,\ e1C,\ \underline{\theta=\{X \mapsto a:S, Y \mapsto b:S\}}}$

3. $[a{:}S \,\square\, \underline{m0} \,\square\,][b{:}S \,\square\, m0 \,\square\,]$
   $[+(d, +(e, b)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, (r, 0) \,\square\, (e1, 0)]$
   $[+(d, +(b, e)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, (r, 0) \,\square\, (e1, 0)]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_2 \,\triangle\, empty \Vdash_{ME}$

4. $[b{:}S \,\square\, \underline{m0} \,\square\,]$
   $[+(d, +(e, b)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, (r, 0) \,\square\, (e1, 0)]$
   $[+(d, +(b, e)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, (r, 0) \,\square\, (e1, 0)]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_3 \,\triangle\, empty \Vdash_{ME}$

5. $[+(d, +(e, b)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, \underline{(r, 0)} \,\square\, (e1, 0)]$
   $[+(d, +(b, e)){:}S \approx Z{:}S \,\square\, (i, 0), n, t, \overline{(r, 0)} \,\square\, (e1, 0)]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_4 \,\triangle\, empty \Vdash_{EL\ \underline{\theta=\{Z \mapsto +(d, +(e, b))\}}}$

6. $[+(d, +(e, b)){:}S \,\square\, \underline{m0} \,\square\,]$
   $[+(d, +(b, e)){:}S \approx +(d, +(e, b)){:}S \,\square\, (i, 0), n, t, (r, 0) \,\square\, (e1, 0)]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_5 \,\triangle\, empty \Vdash_{ME}$

7. $[+(d, +(b, e)){:}S \approx +(d, +(e, b)){:}S \,\square\, (i, 0), n, t, \underline{(r, 0)} \,\square\, (e1, 0)]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_6 \,\triangle\, empty \Vdash_{EL}$

8. $[+(d, +(b, e)){:}S \,\square\, \underline{m0} \,\square\,]$
   $\nabla\, \{X \mapsto a{:}S\} \parallel G_7 \triangle \mathit{empty} \Vdash_{ME}$
9. $\nabla\, \{X \mapsto a{:}S\} \parallel G_8 \triangle \mathit{empty} \Vdash_{AF}$
10. $G_9 \triangle \{X \mapsto a{:}S\}$

We have computed one answer, using the ACU-completed equation *e1C*, whose restriction to the input variables ($\{X \mapsto a\}$) is a solution to our problem. The assignment of an inner variable by A-unification in step 2 generates membership checkings ($a{:}S$, $b{:}S$) in step 3. If more answers are requested, computation resumes from this point. Previous calculus are kept within the structure (G_9).

It must be noticed that a large number of computations may be kept in G_9. This happens because between each step i that we show there have been other steps, not shown, that have modified each G_{i-1}, turning it into G_i. These are partial computations that are still developing and may yield different computed answers.

As previously said transformation rules are like an algorithm that tells us how to apply the calculus rules in chapter 4. We give them a structure to work with, the unification problem, and they use this structure to hold partial results, intermediate variables, evolving the computation. Still there is a long way to the implementation. A lot of details are still not solved, but the base for the implementation is the set of transformation rules.

Chapter 7

Reachability by conditional narrowing

Now we focus on the second part of the calculus, the one that deals with reachability. We have previously defined unification equations and admissible goals. We will now modify our definition for admissible goals to include reachability goals on it, and extend the calculus rules so they can deal with this new type of goals. As previously, we assume that we are working with a Maude module named M. This module has all the declarations for sorts, kinds, operators, memberships, equations, axioms and rules. The new calculus rules will focus on the rules in M, the ones that allow us to rewrite terms.

Conditional narrowing relies on conditional unification, that is, in order to make a narrowing step we previously need to solve a unification problem. Our goal, given a reachability problem $\bigwedge_i s_i{:}S_i \to t_i{:}T_i$, is to find a solution σ (ground or not) such that $\bigwedge_i s_i\sigma{:}S_i \to t_i\sigma{:}T_i$, with the rules R (conditional or unconditional) defined in our module M, modulo axioms and equations in M. Narrowing steps are made with the rules R as we unify the left side of a rule with a subterm at a certain nonvariable position of the left side of some subgoal until we get the solution σ. As for unification, we will imitate narrowing with the following calculus that only applies narrowing, we call it *replacement* here, at position ϵ of terms. First we make some definitions.

Reachability goals are any sequence (understood as conjunction) of subgoals of the form $s{:}S \to t{:}T$, each one meaning that we want to rewrite the term s with sort S to the term t with sort T.

Admissible goals, or simply goals, are now extended to be any sequence of $s{:}S{\to}t{:}T$, $s{:}S{=}t{:}T$, $s{:}S{\approx}t{:}T$, $s{:}S{:=}t{:}T$ and $t{:}T$. From a reachability goal the calculus tries to derive the empty goal. If the calculus succeeds, the substitution used to derive this empty goal, restricted to the input variables, is a computed answer for the reachability goal.

As for unification, any reachability subgoal in our calculus of the form of $s{:}S \to t{:}T$ is equivalent to the admissible goal $s \to t$, $s{=}X_S$, $t{=}Y_T$ (we will use the equivalent writing $s \to t$, $s{:}S$, $t{:}T$.)

7.1 Calculus rules for reachability

Reachability by Conditional Narrowing is achieved using the calculus rules presented in chapter 4, extended with the following calculus rules, under the same conditions for a numerable set of variables, s' and t' kinded variable terms, and c' check for memberships generated by the transformation of s and t:

$[X]$ *reflexivity*

$$\frac{G', s{:}S \to t{:}T, G''}{(G', (c',)s{:}S', t{:}S', G'')\theta}$$

where θ A-unifier of s' and l' kinded variable terms, c' membership checks, $S' \in \mathit{glbSorts}(M, S, T)$.

$[R]$ *replacement*

$$\frac{G', s{:}S \to t{:}T, G''}{G', s{:}S = l{:}S, (c,)s{:}S, r{:}[S] \to t{:}T, G''}$$

where (c)rl $l \to r$ $(if\ c)$ is a fresh variant of a (conditional) rule in R.

$[T]$ *transitivity*

$$\frac{G', s{:}S \to t{:}T, G''}{G', s{:}S \to X{:}[S], X{:}[S] \to t{:}T, G''}$$

$[C]$ *congruence*

$$\frac{G', f(\bar{s}{:}\bar{S}){:}S \to f(\bar{t}{:}\bar{T}){:}T, G''}{G', s_1{:}S_1 \to t_1{:}T_1, ..., s_n{:}S_n \to t_n{:}T_n, f(\bar{s}{:}\bar{S}){:}S, f(\bar{t}{:}\bar{T}){:}T, G''}$$

where the f's are not flattened if f is a binary function.
If they are flattened, we unflatten them using $[E]$.

$[I]$ *imitation*

$$\frac{G', f(\bar{s}{:}\bar{S}){:}S \to X{:}T, G''}{G'\theta, s_i{:}S_i \to X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S, f(\bar{X}{:}\bar{S}){:}T, G''\theta}$$

where $X \notin \mathit{Var}(s), \theta = \{X \mapsto f(\bar{X}{:}\bar{S})\}$, X_i fresh variables.

$[E]$ *equality*

$$\frac{G', s{:}S \to t{:}T, G''}{G', s{:}S{=}X{:}S, X{:}S \to Y{:}T, Y{:}T{=}t{:}T, G''}$$

These rules are a translation of the deduction rules for rewrite theories, using the concepts of *equational conditional rewriting without evaluation of the premise* [Boc93] and *lazy conditional narrowing* [MSH02] for conditional rules. We have only added the *imitation* rule, $[I]$, that allow us to imitate narrowing at non root term positions. All other rules match the corresponding deduction rule.

One important difference with respect to the calculus rules for unification is that only in rule $[X]$, that solves reachability by A-unification, we compute the set $glbSorts(M, S, T)$ and use it. The other rules are not sort-decreasing, so there is no need to do this. In fact, in rule $[T]$ the intermediate variable X gets kind $[S]$ because s can be rewritten to any term within the kind of s, and if this term is rewritten to t, s and t with right sorts, then we may apply the transitivity rule for reachability.

Flattening means representing a term whose root function is associative in a special form, allowed by Maude. For instance, a binary term like $f(a, f(b, c))$, where f is associative, can be represented in Maude as a flattened term $f(a, b, c)$. We use rule $[C]$ only if we have a term written in binary form. In the other case, rule $[E]$ will generate a unification subgoal $f(a, b, c) = X$, which by rule $[u]$ will become $f(a, b, c) \approx X$, and then rule $[i]$ will return us all the unflattened (binary) decompositions for the term, allowing us then to use rule $[C]$.

7.2 Example

In this ACU-coherence completed module:

```
mod R is
  sort S .
  ops a, b, c, d, e : -> S .
  var Z : S .
  op f(_,_) : S S -> S [comm assoc] .
  op g(_,_) : S S -> S .
  rl g(a, b) => c [label r1] .
  eq f(c, d) =  e [label e1] .
  eq f(c, d, Z) =  f(e, Z) [label e1C] .
endm
```

we consider the reachability goal $f(d, g(X, b){:})S \to e{:}S$. The derivation, where the substitution id is assumed when none is shown and some sorts are omitted to make reading the derivation easier, is:

1. $\underline{f(d, g(X, b)) \to e} \rightsquigarrow_{[E]}$
2. $\underline{f(d, g(X, b)) = V}, V \to W, W = e \rightsquigarrow_{[u]}$
3. $\underline{f(d, g(X, b)){:}S \approx Y{:}S}, V \approx Y, V \to W, W = e \rightsquigarrow_{[r],\ \theta=\{Y \mapsto f(d,g(X,b)){:}S\}}$
4. $\underline{f(d, g(X, b)){:}S}, f(d, g(X, b)){:}S, V \approx f(d, g(X, b)), V \to W, W = e \rightsquigarrow_{[m1]}$
5. $\underline{f(d, g(X, b)){:}S}, V \approx f(d, g(X, b)), V \to W, W = e \rightsquigarrow_{[m1]}$
6. $\underline{V{:}S \approx f(d, g(X, b)){:}S}, V \to W, W = e \rightsquigarrow_{[r],\ \theta=\{V \mapsto f(d,g(X,b)){:}S\}}$
7. $\underline{f(d, g(X, b)){:}S}, f(d, g(X, b)){:}S, f(d, g(X, b)) \to W, W = e \rightsquigarrow_{[m1]}$
8. $\underline{f(d, g(X, b)){:}S}, f(d, g(X, b)) \to W, W = e \rightsquigarrow_{[m1]}$
9. $\underline{f(d, g(X, b)){:}S \to W{:}S}, W = e \rightsquigarrow_{[I],\ \theta=\{W \mapsto f(W_1{:}S, W_2{:}S)\}}$
10. $\underline{d{:}S \to W_1{:}S}, g(X, b) \to W_2, f(W_1, W_2) = e \rightsquigarrow_{[X],\ \theta=\{W_1 \mapsto d{:}S\}}$
11. $\underline{d{:}S}, d{:}S, g(X, b) \to W_2, f(d, W_2) = e \rightsquigarrow_{[m1]}$
12. $\underline{d{:}S}, g(X, b) \to W_2, f(d, W_2) = e \rightsquigarrow_{[m1]}$
13. $\underline{g(X, b){:}S \to W_2{:}S}, f(d, W_2) = e \rightsquigarrow_{[R],r1}$
14. $g(X, b){=}g(a, b), \underline{g(X{:}S, b){:}S}, c{:}[S] \to W_2{:}S, f(d, W_2){=}e \rightsquigarrow_{[m1],\mathit{leastSort}(g(X:S,b))=S}$
15. $\underline{g(X, b) = g(a, b)}, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[u]}$
16. $\underline{g(a, b){:}S \approx Z{:}S}, g(X, b) \approx Z, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[r],\ \theta=\{Z \mapsto g(a,b)){:}S\}}$
17. $\underline{g(a, b){:}S}, g(a, b){:}S, g(X, b) \approx g(a, b), c{:}[S] \to W_2{:}S, f(d, W_2) = e$
    $\rightsquigarrow_{[m1],\ \mathit{leastSort}(g(a,b))=S}$
18. $\underline{g(a, b){:}S}, g(X, b) \approx g(a, b), c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[m1],\ \mathit{leastSort}(g(a,b))=S}$
19. $\underline{g(X{:}S, b){:}S \approx g(a, b){:}S}, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[r],\ \theta=\{\underline{X \mapsto a{:}S}\}}$
20. $\underline{a{:}S}, g(a, b){:}S, g(a, b){:}S, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[m1]}$
21. $\underline{g(a, b){:}S}, g(a, b){:}S, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[m1],\ \mathit{leastSort}(g(a,b))=S}$
22. $\underline{g(a, b){:}S}, c{:}[S] \to W_2{:}S, f(d, W_2) = e \rightsquigarrow_{[m1],\ \mathit{leastSort}(g(a,b))=S}$
23. $\underline{c{:}[S] \to W_2{:}S}, f(d, W_2) = e \rightsquigarrow_{[r],\ \theta=\{W_2 \mapsto c{:}S\}}$
24. $\underline{c{:}[S]}, c{:}S, f(d, c) = e \rightsquigarrow_{[m1],\ \mathit{leastSort}(c)=S}$

25. $\underline{c{:}S}, f(d,c) = e \rightsquigarrow_{[m1],\ leastSort(c)=S}$
26. $\underline{f(d,c){:}S = e{:}S} \rightsquigarrow_{[u]}$
27. $f(d,c) \approx U, \underline{e{:}S \approx U{:}S} \rightsquigarrow_{[r],\ \theta=\{U\mapsto e:S\}}$
28. $f(d,c) \approx e, \underline{e{:}S}, e{:}S \rightsquigarrow_{[m1]}$
29. $f(d,c) \approx e, \underline{e{:}S} \rightsquigarrow_{[m1]}$
30. $\underline{f(d,c){:}S \approx e{:}S} \rightsquigarrow_{[n],[e1]}$
31. $\underline{e{:}S}, e{:}S, e{:}S \approx e{:}S \rightsquigarrow_{[m1]}$
32. $\underline{e{:}S}, e{:}S \approx e{:}S \rightsquigarrow_{[m1]}$
33. $\underline{e{:}S \approx e{:}S} \rightsquigarrow_{[r]}$
34. $\underline{e{:}S}, e{:}S \rightsquigarrow_{[m1]}$
35. $\underline{e{:}S} \rightsquigarrow_{[m1]}$
36. $\square$

We have computed the answer $\theta{=}\{X \mapsto a{:}S\}$, which is a solution to our reachability problem. In step 20 the check for a being of type S is caused by the assignment by A-unification of the value a to the inner variable $X{:}S$ in $g(X{:}S, b)$.

Chapter 8

Correctness of the calculus for reachability

We prove correctness of the calculus for reachability with respect to normalized idempotent substitutions for the executable rewrite theory $\mathcal{R} = (\Sigma, \mathcal{E}, R)$.

8.1 Soundness

We prove that given a reachability goal G, if $G \rightarrow_{\sigma}^{*} \square$ then $G\sigma$ can be derived, so σ is a solution for G. Soundness of the reachability calculus is proved by induction on the length of the derivation. Recall that all calculus rules always check correct typings on the premises. We transform any goal $(s{:}S \rightarrow t{:}T)$ into $(s \rightarrow t,\ s{:}S,\ t{:}T)$, but we may use both writings for simplicity. By our previous proof of soundness, we know that if we compute a solution σ for $s{:}S{=}t{:}T$ we can derive $s\sigma{=}t\sigma, s\sigma{:}S, t\sigma{:}T$ using the deduction rules for MEL. We assume that $G \equiv g, G'$ and the last calculus rule has been applied on g.

Base step: proofs with length one. We have a goal with one element. The only inference rule that deletes rewritings without creating new ones is $[X]$:

$[X]$ *reflexivity*

$$\frac{s{:}S \rightarrow t{:}T}{((c'), s{:}S', t{:}S')\theta}$$

where θ A-unifier of s' and l' kinded variable terms, c' membership checks, $S' \in glbSorts(M, S, T)$.

If σ is an answer computed by our unification calculus for $c'\theta$, $s\theta{:}S'$ and $t\theta{:}S'$, then $s\theta\sigma, t\theta\sigma$ are correct terms, with correct typing in the instantiated variables, and

$s\theta\sigma{:}S', t\theta\sigma{:}S'$. $s\theta\sigma{:}S'$, $t\theta\sigma{:}S'$, $S' \leq S$ and $S' \leq T$ implies $s\theta\sigma{:}S$, $t\theta\sigma{:}T$. $s\theta =_A t\theta$ implies $s\theta\sigma =_A t\theta\sigma$. By reflexivity, $s\theta\sigma \to s\theta\sigma$. Then, by equality , $\theta\sigma$ is a solution for $s{:}S \to t{:}T$.

□

Induction step: We assume that if a derivation of □ from a goal G, with length n or less, provides a substitution σ, then $G\sigma$ is derivable, that is, σ is an answer of G. We have to prove that this property holds for derivations with length $n+1$:. We assume $G \equiv g, G'$ (G' may be empty), and check all possible calculus rules applied to g:

[X] *reflexivity*

$$\frac{s{:}S \to t{:}T, G'}{((c',)s{:}S', t{:}S', G')\theta}$$

where θ A-unifier of s' and l' kinded variable terms, c' membership checks, $S' \in glbSorts(M, S, T)$.

As before, if σ is the computed answer, then $\theta\sigma$ is a solution for $s{:}S \to t{:}T$. By I.H., $\theta\sigma$ is also a solution for G', so $\theta\sigma$ is a solution for G.

□

[R] *replacement*

$$\frac{s{:}S \to t{:}T, G'}{s{:}S = l{:}S, (c,)s{:}S, r{:}[S] \to t{:}T, G'}$$

where (c)rl $l \to r$ $(if\ c)$ is a fresh variant of a (conditional) rule in R.

If σ is a computed answer for $s{:}S = l{:}S$, $s{:}S$, c, $r{:}[S] \to t{:}T$ and G', $c\sigma$ is derivable, by I.H. so we derive $l\sigma \to r\sigma$, by replacement. By I.H. $r\sigma \to t\sigma$ is also derivable. Then, by transitivity, $s\sigma \to t\sigma$.
$r\sigma \to t$, $r\sigma{:}[S]$ and $t\sigma{:}T$ are derivable by I.H, and also $s\sigma{:}S$ and $G'\sigma$ are.

Putting all together, we derive $s\sigma \to t\sigma$, $s\sigma{:}S$, $t\sigma{:}T$ and $G'\sigma$.

It is important to remember, again, that ACU-coherence completion allows A-unification of the ACU-coherence completed version of the left term of the equation, l, with the whole term s whenever the left term l can be A-unified with some subterm of a recombination of s.

□

[T] *transitivity*

$$\frac{s{:}S \to t{:}T, G'}{s{:}S \to X{:}[S], X{:}[S] \to t{:}T, G'}$$

If σ is the computed answer, by I.H. we can derive $s \to X$ and $X \to t$ with correct typing, $s{:}S$ and $t{:}T$, as before. Then, by transitivity, we can derive $s \to t$.

□

[I] *imitation*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \to X{:}T, G'}{s_i{:}S_i \to X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S, f(\bar{X}{:}\bar{S}){:}T, G'\theta}$$

where $X \notin Var(s), \theta = \{X \mapsto f(\bar{X}{:}\bar{S})\}$, X_i fresh variables.

$\sigma \equiv \theta\sigma'$, σ' computed answer for $s_i{:}S_i \to X_i{:}S_i, f(\bar{s}{:}\bar{S}){:}S, f(\bar{X}{:}\bar{S}){:}T$ and $G'\theta$.
As $f(\bar{s}\sigma) \equiv f(\bar{s}\theta\sigma') \equiv f(\bar{s}\sigma')$ $(X \notin Var(s))$, by I.H. we can derive $f(\bar{s}\sigma{:}\bar{S}){:}S$.
$X\sigma \equiv X\theta\sigma' \equiv f(\bar{X}\sigma')$ so, by I.H., we can derive $X\sigma{:}T$.
By I.H. we derive $s_i\sigma{:}S_i \to X_i\sigma{:}S_i$. Then, by congruence, $f(\bar{s}\sigma{:}\bar{S}){:}S \to^* f(\bar{X}\sigma') \equiv X\sigma$.
So we can derive $g\sigma$, and σ' answer of $G'\theta$ implies can derive $G'\theta\sigma'$, that is, we can also derive $G'\sigma$.

□

[C] *congruence*

$$\frac{f(\bar{s}{:}\bar{S}){:}S \to f(\bar{t}{:}\bar{T}){:}T, G'}{s_1{:}S_1 \to t_1{:}T_1, ..., s_n{:}S_n \to t_n{:}T_n, f(\bar{s}{:}\bar{S}){:}S, f(\bar{t}{:}\bar{T}){:}T, G'}$$

where the f's are not flattened if f is a binary function.

If σ is a computed solution, then by I.H. we derive $f(\bar{s}\sigma{:}\bar{S}){:}S$, $f(\bar{t}\sigma{:}\bar{T}){:}T$, $s_i\sigma \to t_i\sigma$, $s_i\sigma{:}S_i$, $t_i\sigma{:}T_i$, as before. By congruence we derive $f(\bar{s}\sigma) \to f(\bar{t}\sigma)$.

□

[E] *equality*

$$\frac{s{:}S \to t{:}T, G'}{s{:}S{=}X{:}S, X{:}S \to Y{:}T, Y{:}T{=}t{:}T, G'}$$

If σ is a computed solution then $s\sigma =_{E \cup A} X\sigma, s\sigma{:}S, X\sigma{:}S, Y\sigma =_{E \cup A} t\sigma, Y\sigma{:}T, t\sigma{:}T$, and we can derive $X\sigma \to Y\sigma$. Then, by equality, we derive $s\sigma \to t\sigma$ and, by I.H. we derive $G'\sigma$.

□

8.2 Completeness

We prove that if θ is a normalized idempotent answer of G, then there is σ normalized idempotent, with $\theta \ll_{\mathcal{E}} \sigma$, such that $G \rightsquigarrow^*_\sigma \square$. Completeness of the calculus with respect to a normalized, idempotent answer θ is proved by induction on the length of deductions in $\mathcal{R} = (\Sigma, E \cup A, R)$, looking at the last deduction rule used. We prove that we can compute a normalized, idempotent answer σ such that $\theta \ll_{\mathcal{E}} \sigma$. We omit memberships when working at the kind level:

Base step:

(Reflexivity)

$$\frac{t \in T_\Sigma(X)_k}{(\forall X)t \to t}$$

We have derived $s\theta \to t\theta$ because $s\theta \equiv t\theta$ and $t\theta \in T_\Sigma(X)$. Then $s\theta =_A t\theta$. $s\theta =_A t\theta$ allows the inference $s\theta \to t\theta$. Any instantiated variable x_i in θ must have correct type S_i, that is, we have derived the correct type for it. By I.H we can compute some σ, with $\theta \ll_{\mathcal{E}} \sigma$, answer of $s' =_A t', \bigwedge x_i{:}S_i$, that is, if we drop sorts on inner variables of s and t we get a more general answer for s' and t'. σ being more general that ρ means that the instantiated variables in σ are a subset of those in θ. Then all the instantiated variables in σ have correct type and $s\sigma{:}kind(M,s) =_A t\sigma{:}kind(M,t)$.
$s{:}kind(M,s) \approx t{:}kind(M,t) \rightsquigarrow_{[r],\sigma} s\sigma{:}kind(M,s), t\sigma{:}kind(M,t) \rightsquigarrow^* \square$.

□

Induction step:

(Equality)

$$\frac{(\forall X)u \to u',\ \mathcal{E} \vdash (\forall X)t = u,\ \mathcal{E} \vdash (\forall X)u' = t'}{(\forall X)t \to t'}$$

We have found a solution $\Theta{=}\{\theta, X{\mapsto}u, Y{\mapsto}u'\}$ for the problem $X{\to}Y, t{=}X, Y{=}t'$. Then we have kept θ only. If θ is a normalized idempotent answer for $t \to t'$, as X and Y are fresh variables Θ is also idempotent.
$u{\downarrow}{=}u, u'{\downarrow}{=}u', u{\to}u' \Rightarrow u{\downarrow}{\to}u'{\downarrow}$. $t\theta{=}u{=}u{\downarrow}, t'\theta{=}u'{=}u'{\downarrow}, u{\downarrow}{\to}u'{\downarrow} \Rightarrow t\theta{\to}t'\theta$.
So $\Theta{=}\{\theta, X{\mapsto}u{\downarrow}, Y{\mapsto}u'{\downarrow}\}$ is a normalized idempotent solution for $X{\to}Y, t{=}X, Y{=}t'$.
By I.H. there is a normalized substitution $\Sigma{=}\{\sigma, X{\mapsto}u_1, Y{\mapsto}u'_1\}$ with $\Theta \ll_{\mathcal{E}} \Sigma$ such that we can compute $X{\to}Y, t{=}X, Y{=}t' \rightsquigarrow^*_\Sigma \square$.
Now, applying rule $[E]$, σ is a computed answer for $t \to t'$.

□

(Transitivity)

$$\frac{(\forall X)t_1 \to t_2,\ (\forall X)t_2 \to t_3}{(\forall X)t_1 \to t_3}$$

We have found a solution $\Theta{=}\{\theta, X{\mapsto}t_2\}$ for the problem $t_1{\to}X, X{\to}t_3$. Then we have kept θ only. If θ is a normalized idempotent answer for $t \to t'$, as X is a fresh variable Θ is also idempotent.
$t_2{\downarrow}{=}t_2, t_1{\to}t_2 \Rightarrow t_1{\to}t_2{\downarrow}.\ t_2{\downarrow}{=}t_2, t_2{\to}t_3 \Rightarrow t_2{\downarrow}{\to}t_3$.
So $\Theta{=}\{\theta, X{\mapsto}t_2{\downarrow}\}$ is a normalized idempotent solution for $t_1{\to}X, X{\to}t_3$. By I.H. there is a normalized substitution $\Sigma{=}\{\sigma, X{\mapsto}t_2'\}$ with $\Theta \ll_{\mathcal{E}} \Sigma$ such that we can compute $t_1{\to}X, X{\to}t_3 \rightsquigarrow^*_\Sigma \square$.
Now, applying rule $[T]$, σ is a computed answer for $t_1 \to t_3$.

□

(Congruence)

$$\frac{f \in \Sigma_{k_1 \cdots k_n, k}\ (\forall X)t_i \to t_i'\ \ t_i, t_i' {\in} T_\Sigma(X)_{k_i}, 1{\leq}i{\leq}n}{(\forall X)f(t_1, \ldots, t_n) \to f(t_1', \ldots, t_n')}$$

From $t_i\theta \to t_i'\theta$ we derive $f(t_1', \ldots, t_n')\theta \to f(t_1, \ldots, t_n)\theta$. By I.H. there is σ, with $\theta{\ll_{\mathcal{E}}}\sigma$, such that σ is a computed solution for $t_i{\to}t_i'$. σ is the desired answer:
$f(t_1, \ldots, t_n){\to}f(t_1', \ldots, t_n'){\rightsquigarrow_{[d]}}t_1{\to}t_1', \ldots, t_n{\to}t_n' \overset{I.H.}{\rightsquigarrow}_\sigma \square$.

□

(Replacement)

$$\frac{\begin{array}{c}(\lambda : (\forall X)\, l \to r \ \texttt{if} \bigwedge_i p_i{=}q_i \wedge \bigwedge_j w_j{:}s_j \wedge \bigwedge_k l_k \to r_k){\in}R \\ \theta : X \to T_\Sigma(Y) \quad \bigwedge_i \mathcal{E} \vdash (\forall Y)p_i\theta{=}q_i\theta \quad \bigwedge_j \mathcal{E} \vdash (\forall Y)w_j\theta{:}s_j \quad \bigwedge_k (\forall Y)l_k\theta \to r_k\theta\end{array}}{(\forall Y)l\theta \to r\theta}$$

θ is a solution for the goal $l \to r$. By I.H. there is σ such that $\theta \ll_{\mathcal{E}} \sigma$ and σ is a computed answer for all conditions (we call them c). then:
$l{\to}r \rightsquigarrow_{[R],l\to r} c, r{\to}r \overset{I.H.}{\rightsquigarrow}_\sigma r\sigma{\to}r\sigma \rightsquigarrow_{[X]} \square$.

□

Chapter 9

Transformations for reachability

Again we must turn the previous calculus into a structure and a set of transformations where we can put all the knowledge we acquire, as the computation takes place. The important point is that the structure and the transformations are extensions of the previous ones. Both calculus, for unification and for reachability, will be intertwined, developing at the same time.

Taking as reference the structure and transformations rules for unification shown in chapter 7, we enhance them, allowing also subgoals with structure $G \,\square\, T \,\square\, R$, where:

- G is the actual subgoal. Now it can be rewriting ($\rightarrow$), unification, matching, equation or membership.

- R is a queue/set of inference rules for rewriting. Each element holds a rule from the calculus for reachability and a number for the A-unifier to request for that rule, if needed.

- T is a queue/set of rules in M. This list is used by calculus rule R.

As an example, the following structure:

$$[V \approx f(d, g(X, b)) \,\square\, (r, 0) \,\square][V \rightarrow W \,\square\, (X, 0) \,\square][W = e \,\square\, u \,\square] \,\nabla\, id \,\triangle\, empty$$

has one goal with three subgoals. The first one is a rewriting problem for unification $V \approx f(d, g(X, b))$, the second one is a reachability problem $V \rightarrow W$, where we can only apply rule X with A-unifier 0, and the third one is a unification problem $W = e$. The substitution found so far is the initial one id, and the set of found answers is $empty$.

9.1 Transformation rules for reachability

We add the following rules for problem transformation to the previous ones:

RX RefleXivity

$$[s{:}S \to t{:}T \,\square\, (X,n), \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{RX}$$
$$\bar{G} \parallel [s{:}S \to t{:}T \,\square\, \bar{r}, (X, n{+}1) \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma$$
$$(\parallel [X_i\theta{:}S_i \,\square\, R \,\square\, M]_{i=1}^{n}\ [s\theta{:}S'_j \,\square\, R \,\square\, M]\ \bar{g}\theta \,\nabla\, \sigma\theta)_{j=1}^{m} \,\triangle\, A$$

where $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$, and $\theta = n^{th}$ A-unifier of s' and t', $c'{=}\{X_i{:}S_i\}_{i=1}^{n}$.

$$[s{:}S \to t{:}T \,\square\, (X,n), \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{RX}$$
$$\bar{G} \parallel [s{:}S \to t{:}T \,\square\, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma$$

if no $\theta = n^{th}$ A-unifier of s' and t' exists.

RE Replacement

$$[s{:}S \to t{:}T \,\square\, R, \bar{r} \,\square\, r_1, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{RE}$$
$$\bar{G} \parallel [s{:}S \to t{:}T \,\square\, \bar{r}, R \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel$$
$$\{[s{:}S{=}l{:}S \,\square\, R \,\square\, E](\{[c_i \,\Diamond\, \ldots]\}_{i=1}^{n}\{[(s{:}S) \,\Diamond\, R \,\Diamond\, M]\{[r{:}[S] \to t{:}T \,\Diamond\, R \,\Diamond\, T]\}^{n+3}\}\ \bar{g}$$
$$\nabla\, \sigma \,\triangle\, A$$

where (c)rl $l \to r$ $(if\ c)$ fresh rule from R.

$$[s{:}S \to t{:}T \,\square\, R, \bar{r} \,\square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{RE} [s{:}S \to t{:}T \,\square\, \bar{r} \,\square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A$$

TR TRansitivity

$$[s{:}S \to t{:}T \,\square\, T, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{TR}$$
$$\bar{G} \parallel [s{:}S \to t{:}T \square \bar{r} \square \bar{t}]\bar{g} \nabla \sigma \parallel \{[s{:}S \to Z{:}[S] \square R \square T]\{[Z{:}[S] \to t{:}T \Diamond R \Diamond T]\}\}\bar{g} \nabla \sigma \triangle A$$

$Z{:}[S]$ fresh variable.

IM IMitation

$$[f(\bar{s}{:}\bar{S}){:}S \to X{:}T \,\square\, I, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{IM}$$
$$\bar{G} \parallel [f(\bar{s}{:}\bar{S}){:}S \to X{:}T \,\square\, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma$$
$$\parallel \{[s_i{:}S_i \to X_i{:}S_i \,\square\, R \square T])_{i=1}^{n}\{[f(\bar{s}){:}S \,\Diamond\, R \,\Diamond\, M][f(\bar{X}){:}T \,\Diamond\, R \,\Diamond\, M]\}\}\ \bar{g}\theta \,\nabla\, \sigma\theta \,\triangle\, A$$

where $X \notin Var(s), \theta = \{X \mapsto f(\bar{X}{:}\bar{S})\}$, X_i fresh variables.

CO COngruence

$$[f(\bar{s}{:}\bar{S}){:}S \to f(\bar{t}{:}\bar{T}){:}T \,\square\, C, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{CO}$$
$$\bar{G} \parallel [f(\bar{s}{:}\bar{S}){:}S \to f(\bar{t}{:}\bar{T}){:}T \,\square\, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma$$
$$\parallel \{[s_i{:}S_i \to t_i{:}T_i \,\square\, R \,\square\, T]_{i=1}^{n}\{[f(\bar{s}{:}\bar{S}){:}S \,\Diamond\, R \,\Diamond\, M][f(\bar{t}{:}\bar{T}){:}T \,\Diamond\, R \,\Diamond\, M]\}\}\ \bar{g} \,\nabla\, \sigma \,\triangle\, A$$

where the f's are not flattened if f is a binary function.

EQ EQuality

$$[s{:}S \to t{:}T \,\square\, eq, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EQ}$$
$$\bar{G} \parallel [s{:}S \to t{:}T \,\square\, \bar{r} \,\square\, \bar{t}]\ \bar{g} \,\nabla\, \sigma$$
$$\parallel \{[s{:}S{=}X{:}S \,\square\, R \,\square\, E]\{X{:}S \to Y{:}T \,\lozenge\, R \,\lozenge\, T]\{Y{:}T{=}t{:}T \,\lozenge\, R \,\lozenge\, E]\}\}\ \bar{g} \,\nabla\, \sigma \,\triangle\, A$$

EL ELimination

$$[s \to t \,\square\, \square\,]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{EL} \bar{G} \,\triangle\, A$$

9.2 Example

We consider the same ACU-coherence completed module we used as an example for the calculus rules for reachability:

```
mod R is
  sort S .
  ops a, b, c, d, e : -> S .
  var Z : S .
  op f(_,_) : S S -> S [comm assoc] .
  op g(_,_) : S S -> S .
  rl g(a, b) => c [label r1] .
  eq f(c, d) =  e [label e1] .
  eq f(c, d, Z) =  f(e, Z) [label e1C] .
endm
```

We also consider the same reachability goal $f(d, g(X, b){:}S) \to e{:}S$. We show the relevant steps of the transformation, omitting memberships since there is only one sort, and also omitting variable assignments in the output substitution if they don't affect input variables:

1. $[\underline{f(d, g(X, b)) \to e} \,\square\, \underline{E}, \bar{t} \,\square\, r1]\ \nabla\ id \,\triangle\, empty \Vdash_{EQ}$
2. $[\underline{f(d, g(X, b)) = V} \,\square\, \underline{u} \,\square]\{[V \to W \,\lozenge\, X \,\lozenge]\{[W = e \,\lozenge\, u \,\lozenge]\}\}\ \nabla\ id \parallel \overline{G_2 \,\triangle\, empty \Vdash_{UN}}$
3. $[\underline{f(d, (g(X, b)) \approx Y} \,\square\, (r, 0), \bar{r} \,\square\, (e1, 0)][V \approx Y \,\square\, (r, 0) \,\square]\{[V \to W \,\lozenge\, X \,\lozenge]$ $\{[W = e \,\lozenge\, u \,\lozenge]\}\}\ \nabla\ \overline{id \parallel G_3} \,\triangle\, empty \Vdash_{EL,\ \theta=\{Y \mapsto f(d, g(X,b))\}}$

4. $[\underline{V \approx f(d, g(X, b))} \,\square\, \underline{(r, 0)}\, \square]\{[V \to W \,\Diamond\, X \,\Diamond]\{[W = e \,\Diamond\, u \,\Diamond]\}\} \,\nabla\, id \parallel$
$\overline{G_4 \triangle \mathit{empty}} \Vdash_{EL,\ \theta=\{V \mapsto f(d,g(X,b))\}}$

5. $\{[f(d, g(X, b)) \to W \,\Diamond\, I, \bar{t}\, \Diamond]\{[W = e \,\Diamond\, u \,\Diamond]\}\} \,\nabla\, id \parallel$
$G_5 \triangle \mathit{empty} \Vdash_{RC}$

6. $[\underline{f(d, g(X, b)) \to W} \,\square\, \underline{I}, \bar{t}\, \square]\{[W = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, id \parallel$
$\overline{G_6 \triangle \mathit{empty}} \Vdash_{IM,\ \theta=\{W \mapsto f(W_1,W_2)\}}$

7. $[\underline{d \to W_1} \,\square\, \underline{X}, \bar{t}\, \square][g(X, b) \to W_2 \,\square\, R, \bar{t}\, \square\, r1]\{[f(W_1, W_2) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, id \parallel$
$\overline{G_7 \triangle \mathit{empty}} \Vdash_{RX,\ \theta=\{W_1 \mapsto d)\}}$

8. $[\underline{g(X, b) \to W_2} \,\square\, \underline{R}, \bar{t}\, \square\, \underline{r1}]\{[f(d, W_2) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, id \parallel$
$\overline{G_8 \triangle \mathit{empty}} \Vdash_{RE,[r1]}$

9. $[\underline{g(X, b) = g(a, b)} \,\square\, \underline{u}\, \square\,][c \to W_2 \,\square\, X, \bar{t}\, \square\, r1]\{[f(d, W_2) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, id \parallel$
$\overline{G_9 \triangle \mathit{empty}} \Vdash_{UN}$

10. $[g(X, b) \approx Z \,\square\, (r, 0), \bar{r}\, \square\, e1][\underline{g(a, b) \approx Z} \,\square\, \underline{(r, 0)}, \bar{r}\, \square\, e1][c \to W_2 \,\square\, X, \bar{t}\, \square\, r1]$
$\{[f(d, W_2) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, id \parallel \overline{G_{10} \triangle \mathit{empty}} \Vdash_{EL,\ \theta=\{Z \mapsto g(a,b)\}}$

11. $[\underline{g(X{:}S, b) \approx g(a, b)} \square \underline{(r, 0)}, \bar{r} \square e1][c \to W_2 \square X, \bar{t} \square r1]\{[f(d, W_2) = e \Diamond u \Diamond]\} \nabla id \parallel$
$\overline{G_{11} \triangle \mathit{empty}} \Vdash_{EL,\ \theta=\{\underline{X \mapsto a}\}}$

12. $[\underline{c \to W_2} \,\square\, \underline{X}, \bar{t}\, \square\, r1]\{[f(d, W_2) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, \{X \mapsto a\} \parallel$
$\overline{G_{12} \triangle \mathit{empty}} \Vdash_{EL,\ \theta=\{W_2 \mapsto c\}}$

13. $\{[f(d, c) = e \,\Diamond\, u \,\Diamond]\} \,\nabla\, \{X \mapsto a\} \parallel$
$G_{13} \triangle \mathit{empty} \Vdash_{RC}$

14. $[\underline{f(d, c) = e} \,\square\, \underline{u}\, \square] \,\nabla\, \{X \mapsto a\} \parallel$
$\overline{G_{14} \triangle \mathit{empty}} \Vdash_{UN}$

15. $[f(d, c) \approx U \,\square\, n, \bar{r}\, \square\, (e1, 0)][\underline{e \approx U} \,\square\, \underline{(r, 0)}, \bar{t}\, \square\, (e1, 0)] \,\nabla\, \{X \mapsto a\} \parallel$
$\overline{G_{15} \triangle \mathit{empty}} \Vdash_{EL,\ \theta=\{U \mapsto e\}}$

16. $[\underline{f(d, c) \approx e} \,\square\, \underline{n}, \bar{r}\, \square\, \underline{(e1, 0)}] \,\nabla\, \{X \mapsto a\} \parallel$
$\overline{G_{16} \triangle \mathit{empty}} \Vdash_{NA,[e1]}$

17. $[\underline{e \approx e} \,\square\, \underline{(r, 0)}, \bar{r}\, \square\, (e1, 0)] \,\nabla\, \{X \mapsto a\} \parallel$
$\overline{G_{17} \triangle \mathit{empty}} \Vdash_{EL}$

18. $\nabla\, \{X \mapsto a\} \parallel G_{18} \triangle \mathit{empty} \Vdash_{AF}$

19. $G_{19} \triangle \{X \mapsto a\}$

We have found the same answer $\sigma = \{X \mapsto a\}$ to our reachability problem. The A-unification in step 10 generates a membership check $a{:}S$ which has been omitted as previously stated.

Again, G_{17} keeps a large number of computations as we have only shown the interesting steps, omitting many of them.

Chapter 10

Implementation

10.1 Prototype

When translating the previous rules for problem transformation into a Maude's program an almost direct translation of them has been made. The program works in the metalevel, because it needs to take the module we are working with as a parameter. It is a functional module, that is, a functional program where the equations tell us how an initial goal gets transformed. The main loop works as follows: from a set of goals, each one describing a possible partial computation of an answer, we choose one goal. This goal is processed and a set of new goals and/or answers is generated. New goals are enqueued with the previous ones, and new answers are added to the set of found answers.

Maude's module full-maude.maude is loaded previously because it provides several useful metalevel functions, specially `acuCohComplete` for ACU coherence completion of modules.

10.1.1 Structures

ControlStructure

The main structure that holds the whole computation is called `ControlStructure`. Initially it holds one goal, that may have several subgoals:

```
op _,_,_,_,_,_,_,_ : GoalList GoalList GoalList Module Int AnswerSet TermList
                     ControlFlag -> ControlStructure [ctor] .
op _,_ : Goal GoalList -> GoalList [ctor] .
```

The first goal list is used to choose a new goal to perform one step of processing. If the goal may have further processing, we put it on the second goal list. New goals, or new answers, generated during this processing are returned in the third goal list and later moved into the second one. When the first list is empty, we switch places

with the second one. In this way we avoid recursive calls for insertion of goals at the end of a queue of goals. `Module` is the Maude module used for the unification or reachability problem. `Int` holds a number that serves for creating fresh variables, variables not present anywhere in a goal. The same fresh variable may appear in different goals. `AnswerSet` holds the answers found so far. `TermList` holds the set of variables that are present in the original goal. It is used to discard assignments to intermediate variables when inserting a new answer in `AnswerSet`. `ControlFlag` is used as a signal for operator `developGoals` when there are new goals generated that must be added to the second goal list. If all problem lists get empty, there can be no further processing. Found answers, if there is any, will be stored in `AnswerSet`.

Goals

Each goal is made up of several subgoals:

```
op _,_,_,_ : SubgoalList SubgoalList Substitution Int -> Goal [ctor] .
op _,_ : Subgoal SubgoalList -> SubgoalList [ctor] .
```

Again, we keep two subgoal lists. We choose a subgoal from the first list for processing, and we put it back on the second list if it can have further processing. When the first list gets empty, it switches places with the second one. `Substitution` holds the partial answer computed so far for this goal. If both subgoal lists get empty, then `Substitution` holds an answer to the original goal. Each subgoal of a goal is uniquely identified by an integer called `NID`. `Int` holds the number to use as `NID` if we generate a new subgoal for this goal. Different goals may have subgoals with same `NID`, but each subgoal in a goal has different `NID` .

Subgoals

A subgoal may have different form depending on whether it is a membership subgoal or any other kind of subgoal:

```
op _;_,_,_,_,_ : MbSubgoal TermType CalcRules ControlPairs Int Ndeps
                 -> Subgoal [ctor] .
op _;_,_,_,_,_,_,_ : UnEqMaRe  TermType TermType TypeSet CalcRules
                     ControlPairs Int Ndeps -> Subgoal [ctor] .
op m : -> MbSubgoal .          *** Membership
ops u n x : -> UnEqMa .        *** Unification, Unif. by rewriting, Matching
op r : -> ReSubgoal .          *** Reachability
op _-_ : Term Type -> TermType [ctor] .
subsort UnEqMa  ReSubgoal < UnEqMaRe < UnEqMbMaRe .
subsort MbSubgoal < UnEqMbMaRe .
```

The first field identifies the type of the subgoal. A membership subgoal has one `TermType` field, meaning that the term must be of this type. The other subgoals

have all the same form, we only distinguish reachability subgoals from the others for processing purposes. They have two `TermType` fields, that are the terms that we want to process, together with the type that each term must have. They also have a `TypeSet` field, where we we keep $glbSorts(M, S, T)$ as part of the subgoal, saving us computing time because it is constant for each subgoal. The other fields are common to all subgoals. The syntax for `Type`, `Term` and `TypeSet` can be found in the `metalevel` section of the file `prelude.maude` (we are working at the metalevel). `CalcRules` holds the calculus rules that can be applied to the subgoal. `ControlPairs` holds the rules, equations or memberships that can be applied to the subgoal. `Int` is the `NID` of the subgoal. The structure of nested braces in transformation rules is implemented with the set of dependencies `Ndeps`, a set of positive integers (`NID`s), plus the number zero. The only subgoals that can be chosen for development are those whose set of dependencies `Ndeps` has only one element, the number zero, meaning that this subgoal doesn't depend on any other one. This is the equivalent to the $\Box$ symbol on transformation rules. The other subgoals are considered to contain $\Diamond$ symbols. In each goal we have an item `NNID` that holds the next `NID` number to be used when creating a new subgoal. If a subgoal depends on other subgoals, the `NID`s of all the subgoals it depends on are included in the subgoal's `Ndeps` field, together with the number zero. This is the equivalent to braces in transformation rules. Each time a subgoal is solved we remove its `NID` from the `Ndeps` field of all other subgoals in the goal. In this way a subgoal that depends on several subgoals, a $\Diamond$ subgoal, can develop when the `NID`s it holds for that subgoals are removed from its `Ndeps` field, and this `NDEP` field holds only the number zero, that is, when the subgoal has become a $\Box$ subgoal, which is the equivalent to the resume computation (**RC**) transformation rule.

10.1.2 Control operators

Now we explain the operators that manage the main loop of the prototype. There are four operators that turn a user's request into a control structure and another four that manage this control structure.

process

```
op process : UnEqMaRe Qid Term Type Term Type Int  -> ControlStructure .
```

This operator is the user's interface to the program and may be called, for instance, in the following way:

```
reduce process(u, 'M, 'N:Nat, 'Nat, '0.Nat, 'Nat, 1) .
```

where `reduce` is Maude's reserved word for finding the canonical form of a term, that is, apply equations from our program until there are no equations left to

apply. The first field can be `u` if we want to unify two terms or `r` if we are asking for reachability. `Qid` is a quoted identifier, in this case `'M`, that corresponds to a previously loaded functional or system module in Maude, in this case it should be called `M`. The following couple of `Term` and `Type` are the two terms and types that we want to unify or check reachability. In the example, the initial goal is to unify, using the previously loaded module `M`, the variable `N:Nat` with the constant `0.Nat` and both must have sort `Nat`. The syntax `N:Nat`, `0.Nat` from Maude means that `N` is a variable with sort `Nat` and `0` is a constant with sort `Nat`. Finally `Int`, in this case `1`, means the number of times that the main loop `developGoals` must be called.

The operator `process` allows us to specify initial goals with only one subgoal, but it can be easily modified to manage term lists and type lists so that initial goals with multiple subgoals can be specified.

generate

```
op generate : UnEqMaRe Qid Term Type Term Type  -> ControlStructure .
```

This operator takes the user input and generates the corresponding control structure for the initial goal. Before generating this structure it calls `preprocess`, which performs two important transformations.

preprocess

```
op preprocess : Qid -> Module .
```

This operator takes the quoted name of a module, which must have been previously loaded into the system, and returns the ACU-coherence completed metalevel version of the module, using the metalevel operators `upModule` and `acuCohComplete`. It also adds a membership for each operator, as stated in chapter 4.

iterate

```
op iterate : ControlStructure Int -> ControlStructure .
```

Once that `process` has generated the initial control structure, it invokes `iterate`, that merely calls the main loop, `developGoals`, `Int` times.

developGoals

```
op developGoals : ControlStructure -> ControlStructure .
```

This is the main loop of the program. It takes a control structure, and if there are there are new goals generated it checks whether each one is a new answer, then it appends it to the list of found answers, or not, then it enqueues the goal in the list of goals to process. If there are not new goals, it chooses one goal and calls `developGoal`.

developGoal

```
op developGoal : Goal ControlStructure -> ControlStructure .
```

This operator processes one goal. If it is a new answer, it appends it to the list of found answers. If it is not, it calls `developSubgoal` and `processResult` to generate new goals.

processResult

```
op processResult : GLMI GoalList GoalList AnswerSet TermList -> ControlStructure .
op _,_,_ : GoalList Module Int -> GLMI [ctor] .
```

This operator generates a new control structure by relocating the new goal list, module and number for fresh variables returned by `developSubgoal` and the rest of parameters coming from the previous control structure. `controlFlag` is set to `addP` and `developGoals` is called to check if there are new goals or answer.

developSubgoal

```
op developSubgoal  : GMI -> GLMI .
op _,_,_ : Goal Module Int -> GMI [ctor] .
```

Given a goal, a module and an integer, the number used for generating the last fresh variable, this module selects an active subgoal, that is with field `Ndeps` equal to `0`, and calls the corresponding processing operator, depending on the value of the first field of the subgoal, the one that tells us what type of subgoal it is. It returns a list of new goals, that might be empty, the module and an integer, which is an update for the number used for generating fresh variables.

10.1.3 Subgoal operators

For each type of subgoal we have have a corresponding operator:

```
op processM    : TermType CalcRules ControlPairs Int Goal Module Int -> GLMI .
op processR    : GMI -> GLMI .
op processN    : GMI -> GLMI .
op processU    : TermType TermType TypeSet Int Goal Module Int -> GLMI .
op processX    : TermType TermType TypeSet Int Goal Module Int -> GLMI .
```

These operators select which calculus rule to apply to the chosen subgoal. Operators `processU` (unification) and `processX` (matching) can only apply one calculus rule, so they apply it. The other three operators check what calculus rule they have to apply and call the operator that implements the application of that rule.

10.1.4 Reachability operators

```
op processRX   : TermType TermType TypeSet EqCSp Int CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
op processRP   : TermType TermType TypeSet CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
op processCO   : SLI TermType TermType TypeSet CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
op processEQ   : TermType TermType Variable Variable TypeSet CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
op processTY   : TermType TermType Variable TypeSet CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
op processIT   : TlSlII TermType TermType TypeSet CalcRules ControlPairs
                 Int Goal Module -> GLMI .
```

Each one of these operators implement one of the following reachability rules:

```
op rx : -> CalcRuleIndex .           *** reflexivity
op rp : -> CalcRuleSingle .          *** replacement
op co : -> CalcRuleSingle .          *** congruence
op eq : -> CalcRuleSingle .          *** equality
op ty : -> CalcRuleSingle .          *** transitivity for reachability
op it : -> CalcRuleSingle .          *** imitation for reachability
```

where `CalcRuleIndex` is applied to rules that need a number showing which unifier must be asked for.

10.1.5 Unification operators

```
op processIM   : TermType TermType TypeSet TlSuSuIM CalcRules
                 ControlPairs Int Goal -> GLMI .
op processNA   : TermType TermType TypeSet EqCSp CalcRules
                 ControlPairs Int Goal Module Int -> GLMI .
op processRE   : TermType TermType TypeSet EqCSp Int CalcRules
                 ControlPairs Int Goal Module Int -> GLMI .
op processTR   : TermType TermType TypeSet Int Goal Module Int GoalList -> GLMI .
op processDC   : Term TermList TypeSet Int Goal Module -> GoalList .
```

Each one of these operators implement one of the following rules for unification by rewriting:

```
op im : -> CalcRuleSingle .          *** imitation for unification
op na : -> CalcRuleSingle .          *** narrowing
op re : -> CalcRuleIndex .           *** removal of equations
op tr : -> CalcRuleSingle .          *** transitivity for unification
op de : -> CalcRuleSingle .          *** decomposition
```

10.1.6 Membership operators

```
op processM1   : TermType CalcRules ControlPairs Int Goal Module Int -> GLMI .
op processM2   : TermType EqCondition Substitution CalcRules ControlPairs
                 Int Goal Module Int -> GLMI .
```

Each one of these operators implement one of the following rules for membership checking:

```
ops m1 : -> CalcRuleSingle . *** direct membership
ops m2 : -> CalcRuleSingle . *** membership parsing
```

Subject reduction rule (`sr`) and ground term memberships checking rule (`m0`) are directly implemented in operator `processM`.

10.1.7 Examples

Example 1: Unification

Recall the transformation rule for unification:

UN UNification

$$[s{:}S{=}t{:}T \,\square\, u \,\square\,]\ \bar{g} \bigtriangledown \sigma \parallel \bar{G} \bigtriangleup A \Vdash_{UN}$$
$$\bar{G}\ (\parallel [s{:}S'_j{\approx}X_j{:}S'_j \,\square\, R_1 \,\square\, E_1]\ [t{:}S'_j{\approx}X_j{:}S'_j \,\square\, R_2 \,\square\, E_2]\ \bar{g} \bigtriangledown \sigma)_{j=1}^{m} \bigtriangleup A$$

$$X_j \text{ fresh variables, } \{S'_j\}_{j=1}^{m}{=}\mathit{glbSorts}(M, S, T).$$

This rule is implemented by operator `processU` which updates the dependencies for old subgoals (the ones that depended on the unification subgoal will now depend on the newly created subgoals) and calls `processU*`, which in turn calls `processU**` once per each type in $\mathit{glbSorts}(M, S, T)$. `processU**` is declared as:

```
op processU**  : TermType TermType Type Int SubgoalList SubgoalList
                 Substitution Module Int GoalList -> GoalList .
```

Where `GoalList` is the list of new goals (one per type), and the rest of the goal that we are processing ($\bar{g}$ in the rule) can be found on both `subgoalList` fields. The code for `processU*` is:

```
eq processU**((TE1 - TY1), (TE2 - TY2), TY, NID, SL, SL2, SU, MO, NV, GL)
 = SL, (putRules((n ; (TE1 - TY), (newVar(NV, TY) - TY), TY, nilCR, nilCP, NID,
                 0), MO),
     putRules((n ; (TE2 - TY), (newVar(NV, TY) - TY), TY, nilCR, nilCP, s(NID),
              0), MO),
     SL2), SU, (NID + 2), GL .
```

From the original goal $TE1 : TY1 = TE2 : TY2, \bar{g}$ we create a new goal for each type TY. We keep `SL` as first subgoal list, `SU` as substitution, and make `NNID`, the number to use as `NID` when creating a new node, equal to `NID` plus two because we are creating two subgoals that we append to the second subgoal list `SL2`. The first subgoal $TE1 : TY = \#NV{:}TY : TY$ matches the first new subgoal in the transformation rule. The second subgoal $TE2 : TY = \#NV{:}TY : TY$ matches the second new subgoal in the transformation rule. As previously said $\bar{g}$ is the union of subgoal lists `SL` and `SL2`. `putRules` is an auxiliar operator that replaces `nilCR` and `nilCP` with the control rules and control pairs, equation pairs in this case, that may apply on each subgoal. This selection is made based on the form of both terms of the given subgoal. There is a distinction between variables, constants, functions and, as a special case, subgoals with the same root function on both terms. When putting rules for memberships there is also a distinction between ground and non ground terms. When needed, `putRules` may call `putEqPairs` or `putMbPairs`, the real operators that replace `nilCP` with the corresponding memberships or equations. In the case of reachability subgoals, `nilCP` is replaced, when needed, with `getRls(MO)` the whole set of rules, because the replacement rule for reachability tell us to try to unify the left term of the subgoal with the left term of any available rule.

Example 2: Narrowing

This is a more complex example. Recall the main part of the transformation rule for narrowing:

NA NArrowing

$[s{:}S \approx t{:}T \,\square\, n, \bar{r} \,\square\, (e_1, n), \bar{e}]\ \bar{g} \,\nabla\, \sigma \parallel \bar{G} \,\triangle\, A \Vdash_{NA}$
$\bar{G} \parallel [s{:}S \approx t{:}T \,\square\, \bar{r}, n \,\square\, \bar{e}, (e_1, n{+}1)]\ \bar{g} \,\nabla\, \sigma$
$(\parallel \{[c_1\theta \,\square \ldots](\{[c_i\theta \,\Diamond \ldots])_{i=2}^{p}\{[r\theta{:}S'_j \approx t\theta{:}S'_j \,\Diamond\, R \,\Diamond\, E]\} \ldots\}\ \bar{g}\theta \,\nabla\, \sigma\theta)_{j=1}^{m} \,\triangle\, A$

where $e_1 \equiv$ (c)eq $l = r$ $(if\ c)$ is a fresh variant of a (conditional) equation in E,
θ n^{th} A-unifier of s' and l', $c' \cup c{=}\{c_i\}_{i=1}^{p}$, $\{S'_j\}_{j=1}^{m} = glbSorts(M, S, T)$.

The last line of the rule tells us that we must first check the memberships for variables in the A-unifier (s' and l' are kinded versions of s and l) and then the rest of conditions in the equation before applying the body of the rule.

The operator that implements this rule is called `processNA`. It is called by operator `processN` in the following way:

```
eq processN((((n ; (TE1 - TY1), TT2, TS, (na, CRS), EPS, NID, 0), SL), SL2, SU,
                    NNID), MO, NV
 = processNA((TE1 - TY1), TT2, TS, makeEqCSp((te2MAS(TE1)), MO),
            CRS, EPS, NID, SL, SL2, SU, NNID, MO, NV) .
```

That is, before calling `processNA`, two other operators are called. `te2MAS(TE1)` takes as input the term `TE1`, and returns a membership axiom set, where for each variable `V` in `TE1` there is one membership of the form `mb V : S`, where `S` is the sort of `V`. This uniqueness is guaranteed because membership axiom sets are defined idempotent in `prelude.maude`, so any variable appearing several times in the term will generate only one membership axiom. The call to `makeEqCSp` takes as input the membership axiom set and the module and returns two elements. The first one, `EC1` is the part of the conditions (part of the c_i's) corresponding to the variables in `TE1`, that is is a join of the given memberships; the second one is a pair of substitutions named `KS` and `UKS`. Substitution `KS` (for kind substitution) replaces every variable `V:S` appearing in `TE1` with `V:[S]`, that is it gives us the kinded version of `TE1`. Substitution `UKS` (for unkind substitution) does the opposite. Now `processNA` is called, which in turn calls `processNA*`:

```
eq processNA((TE1 - TY1), TT2, TS, (EC1, (KS, UKS)), CRS, EPS, NID, SL, SL2, SU,
              NNID, MO, NV)
 = processNA*((TE1 << KS - TY1), TT2, TS, EC1, UKS, CRS, EPS, NID, SL, SL2, SU,
               NNID, MO, NV) .
```

We just make `TE1` a kinded term by the application (`<<`) of substitution `KS` and call `processNA*`, which makes a distinction based on the chosen equation rule. We see the more general case, which is the one for conditional rules:

```
eq processNA*(TTK, TT2, TS, EC1, UKS, CRS, (((ceq T1 = T2 if EC [AtS] .), NA),
              EPS), NID, SL, SL2, SU, NNID, MO, NV)
 = processNA**(TTK, TT2, TS, EC1, UKS, CRS, (((ceq T1 = T2 if EC [AtS] .), NA),
               EPS),T1, T2, EC, makeEqCSp((te2MAS(T1)), MO), NID, SL, SL2, SU,
               NNID, MO, NV) .
```

The equation is `ceq T1 = T2 if EC [AtS] .`, and the required unifier is number `NA`. `processNA*` just generates the conditions for variables in `T1`, named `EC2`, and the kind-unkind substitutions for `T1`, named `KS2` and `UKS2`, and calls `processNA**`. The previous steps can be made in parallel, but we have chosen to separate each step to make the program easier to read and maintain.

```
eq processNA**((TK1 - TY1), TT2, TS, EC1, UKS, CRS, ((EQ, NA), EPS), T1, T2,
                EC, (EC2, (KS2, UKS2)), NID, SL, SL2, SU, NNID, MO, NV)
 = processNA2(metaDisjointUnify(MO, TK1 =? (T1 << KS2), NV, NA), (TK1 - TY1),
              TT2, TS, T2, EC1, EC2, EC, UKS, UKS2, CRS, ((EQ, NA), EPS), NID,
              SL, SL2, SU, NNID, MO, NV) .
```

Now `processNA**` calls `processNA2` with the result of the metalevel function `metaDisjointUnify` for the already kinded term `TK1` and the kinded left part of the equation (`T1 << KS2`), which returns the pair of A-unifiers number `NA` for the kinded terms, if possible. The first A-unifier must be applied to `TK1` and the other A-unifier to `T1 << KS2` to unify this terms. This happens because

`metaDisjointUnify` assumes that the variables in the first term are different to the variables in the second term, even if some of them have the same name on both terms, so two substitutions must be provided. We see the equation that handles the case when the unifier exists:

```
eq processNA2({SU1, SU2, NV'}, (TK1 - TY1), (TE2 - TY2), TS, T2, EC1, EC2, EC,
              UKS, UKS2, CRS, ((EQ, NA), EPS), NID, SL, SL2, SU, NNID, MO, NV)
 = (SL, ((n ; (TK1 << UKS - TY1), (TE2 - TY2), TS, (na ++ CRS),
             ((EQ, s(NA)) ++ EPS) , NID, 0), SL2), SU, NNID),
    processNA3(condNarrGoal(undoKS(UKS, SU1), undoKS(UKS2, SU2), EC1, EC2, EC,
               NID, (SL, SL2, SU, NNID), MO), (TE2 << undoKS(UKS, SU1)), TS,
               (T2 << undoKS(UKS2, SU2)), MO, nilG), MO, NV' .
```

We have found substitutions `SU1` and `SU2`, and the highest number used for fresh variables on them is `NV'`. The first two lines on the right part of this equation implement the second line of the narrowing rule: one of the new goals is the same as the original one, except that the number of unifier is increased (`s(NA)`) and put at the end (`++`) of the queue of equation pairs (`EPS`). Notice that this modified subgoal is appended to the processed subgoal list `SL2` to allow us to select a new subgoal from `SL` the next time that this goal gets selected for development.

`processNA2` returns an object with sort `GLMI`, that is, a goal list, a module and an integer. The module and the integer (the update of the number for fresh variables) go at the end of the rule. We have seen how the first goal of the goal list is generated. Now `processNA3` constructs the rest of the goal list, that is, it implements the last line of the transformation rule for narrowing, one goal per type in `TS` ($glbSorts(M, S, T)$). One of its parameters is the output of `condNarrGoal`, which takes as input the unkinded versions of substitutions `SU1` and `SU2`, the whole set of conditions `EC1`, `EC2` and `EC`, the `NID` of the previous subgoal, the rest of the goal `SL, SL2, SU, NNID` and the module `MO`, and returns the common conditional part for all new goals (the c_i's), the old part of the goal with the new unifier applied and the dependencies updated, and a number which is used for creating dependencies on the subgoals of the new goals that have not been generated yet. The other parameters of `processNA3` are the right terms of the subgoal (`TE2`) and the equation (`T2`) with the corresponding unkinded substitutions (`undoKS(UKS, SU1)` and `undoKS(UKS2, SU2)`) applied, the module (`MO`), and the empty goal list (`nilG`). Now we see the case of `processNA3` when there is only one type left to process in `TS`:

```
eq processNA3(((SL, SL2, SU, NNID), DEP1), TE2, TY, T2, MO, GL)
 = (SL, putRules((n ; (T2 - TY), (TE2 - TY), TY, nilCR, nilCP, NNID,
                 iniNDEP(DEP1, DEP1)), MO), SL2, SU, s(NNID)), GL .
```

`SL, SL2, SU, NNID` is the conditional part of the new goals (the c_i's) (`SL`), the rest of the subgoals of the old goal (`SL2`), with unification applied and dependencies updated, together with the current partial answer updated (`SU`) and the number to use for new subgoals (`NNID`). `DEP1` is the number to use for new dependencies, `TE2` is $t\theta$, `TY` is the only sort left in $glbSorts(M, S, T)$, `T2` is $r\theta$, `MO` is the module and `GL` is the list of goals created so far.

Now this last goal is created and appended to the goal list `GL`. The first subgoal list is that of the c_i's. In the old subgoal list we append the only subgoal left to create $(r\theta{:}S'_j \approx t\theta{:}S'_j)$ which is `(n ; (T2 - TY), (TE2 - TY), TY, nilCR, nilCP, NNID, iniNDEP(DEP1, DEP1))`. The `NID` for this subgoal is `NNID` and it depends on `DEP1`, the last created c_i. `SU` remains unchanged and `NNID` is updated. It is worth noting, finally, that `condNarrGoal` has previously made the old subgoals in `SL2` that depended on the processed original subgoal depend exactly on this new subgoal with identifier `NNID`, as the transformation rules do.

10.2 Improvements

It is also possible to design a more sophisticated structure in order to avoid some of the redundancy of the implemented program, which may generate and process the same goal many times, achieving better performance through several improvements. We sketch some of them here:

- We identify disjoint parts of the goals, that is, with no common variables, solve them independently and join the results.
- Treat these parts as trees. If any of the parts fails to solve, the whole goal is discarded. Inside each tree we can keep track of the variables and subgoals that have been created by each node. If we instantiate any of the variables in one node, we add new goals to this node applying the substitution to any child of the node that contains the variable. Subgoals that are not modified in the new goals are marked and not developed, as they are developing elsewhere. Applied substitutions are kept on the parent node, avoiding repeated substitutions to be tried. If a substitution is successful and the variable instantiated was not created by the parent node, it is sent to the grandparent node for testing.
- If a subgoal is exhausted, then no further development is possible; we keep it as a fail, if it has no answers, or we keep a list of found answers. If it is a fail, any subgoal that is an instantiation of this subgoal is also a fail. If it's not, we can try these answers on any subgoal that is more general than this one, regardless of searching for other answers. Newly created nodes are checked for matching against this set of ended subgoals, trying to obtain either failure or an initial set of answers.

This whole structure allows us to resume at any time, since new answers don't interfere with found ones.

The structure can be represented in Maude as a control tree. This tree grows or shrinks as the calculus develops. There are three types or nodes alternating

in the tree: *goal*-nodes, *and*-nodes and *or*-nodes. The root of the tree is an *or*-node with no rules on it, and it has one child: a *goal*-node that holds the original *A*-unification problem split into independent *and*-nodes. Instantiation is the only way to create new children for this *or*-node, and this must be done inside the *and*-nodes or below them when they develop. All nodes have an index that identifies them. The index is a unique list of numbers, one per level. Root node has index *nil*. We append at the end of a node's index a different natural number each time it creates a child, and this list of numbers is the child's index.

Now, we examine in more detail the structure and operation of the nodes in this control tree .

10.2.1 *Goal*-nodes

Goal-nodes are intermediate nodes between a grandparent *or*-node and a grandchild *and*-node. Their purpose is to hold the independent *and*-nodes of a goal. If one of the *and*-nodes finds a substitution, it is sent to the grandparent *or*-node to be tried on other goals. If one of the independent *and*-nodes fails, the whole *goal*-node fails and this information is sent to the parent *or*-node. All child *and*-nodes must be solved to consider that the *goal*-node is solved.

10.2.2 *And*-nodes

An *and*-node is a multiple goal control node. It verifies that all goals get unified, developing each goal, which are *or*-nodes, and giving control back to its parent. The *and*-node sends to its parent node, which is always a *goal*-node, the substitution found for any variable or a fail status if there is no such substitution. All child *or*-nodes must be solved to consider that the *and*-node is solved.

When the *and*-node is first created we build a connection graph, where each goal is a node and there is one arc between each pair of nodes for every common variable they share, labeled with the name of the variable. Every unconnected subgraph of the graph is independent. Of course, every ground term is independent, as it has no variables. They are all reduced and if any of them fails, the whole *and*-node fails.

When a new *and*-node is created due to a partial substitution found, the goals that are present in the *and*-node that generates the substitution but are not affected by it, are marked in the new *and*-node. Indeed, they hold a reference to the original goal. These marked goals don't develop, because they are already developing in another *and*-node. They just wait for substitutions or failure. Independent parts of the goal are of course marked. If the new *and* node is created due to a new rule applied, the new goals can be checked against the other sibling *and*-nodes, also trying to avoid duplicate computations.

If one of the subgoals of an *and*-node finds a partial substitution, it increases a counter of possible answers and the grandparent *or*-node tries this substitution, as explained before. If the substitution fails, the grandparent *or*-node sends a message to the *and*-node, and the counter is decreased. If the subgoal gets exhausted, and all the substitutions it has found fail, the subgoal is a failure and the whole *and*-node is also a failure. A goal all whose variables have been created by its parent node stops developing as soon as one instantiation of all these variables succeeds, because we only need to know that the goal is achievable. It is of no interest how we achieve it. All goals created by partial instantiation of these variables are erased, for the same reason.

10.2.3 *Or*-nodes

An *or*-node is a single subgoal control node. It tries different ways for solving one subgoal. It is in charge of all the subtree that lies below itself, each one a different computation to solve this subgoal, creating new goals, deleting others, and allowing the development of them. The *or*-node keeps a queue of calculus rules and a queue of equations, rules or memberships that can be applied to generate new subgoals, along with the A-unifiers already applied in the computations. It also keeps another queue of subgoals (*goal*-nodes), which is the actual node subtree.

Each *or*-node holds a list of computed A-unifiers. If an *or*-node ends its computation with a set of computed A-unifiers, it is stored in a list that holds these possible A-unifiers, so that this computation is not repeated. If it ends without answer it is stored in a list of failing goals. These lists can be managed to avoid excessive size. The most obvious way is keeping only a fixed number of elements, which are the most frequently requested ones. When inserting a new goal, if the list is full, we delete one of the least frequently requested ones.

When it gets control from its parent the node develops its children or creates new ones. The node develops its children one at a time until it ends the developing turn or a new substitution is found. Previous substitutions are kept in the *or*-node. The node tries to create a new child by choosing a calculus rule (remember that we have added a special rule $[m0]$) and attempting to apply the rule according to the transformation rules. Some rules can be used only once, so once used they disappear from the list, but other may be used with different A-unifiers. If the rule is narrowing ($[n]$) or membership ($[m2]$), then an equation or membership is chosen and asked for a new A-unifier. If the rule is replacement ($[R]$), a rewrite rule is chosen and the corresponding transformation rule is applied. Equations and memberships that fail to get a new A-unifier are deleted from the queue as they are no longer needed. If the A-unification succeeds, the new *goal*-node, consisting of one or several *and*-nodes depending on the connection analysis of it, is created. On every *and*-node, each new goal gets a list of calculus rules, and a list of rules,

equations or memberships that may apply to it. Goals that are already developing or are in the list of finished goals, having a list of A-unifiers are marked as they have a different treatment. If any goal is in the list of failing goals, the new *goal*-node is discarded. If a goal is a renaming of a parent's goal, the *and*-node is discarded. We have arrived here by circular reasoning. Ground goals are reduced, and if any of them fails the *goal*-node is discarded. Goals are ordered, we prefer those that already have A-unifiers, trying to generate substitutions as soon as possible.

Development of children is as follows: a goal is chosen. Control is given to it, to perform development. If it fails, it is deleted, and control is given back to the parent node. If the *or*-node has no children left, and it cannot create any goal, then a fail status is sent back to the parent, the goal can not be unified. The parent *and*-node is a fail, and all marked nodes that were waiting for this computation also fail. The subnode is added to the list of fail computations. If the chosen goal returns a substitution, control is returned to the parent node, together with the substitution found. If the subgoal returns a working status, meaning that the tree it controls has new nodes, but no substitution has been found yet, control is given back to the parent node returning also a working status, allowing the development of other branches of the control tree.

The control tree must be fair to nodes, calculus rules, rewrite rules, equations and memberships. This can by ensured if we develop the control tree in a breadth first way, as the set of A-unifiers for two terms returned by Maude is always finite, as well as the sets of calculus rules, rewrite rules, equations and memberships.

It is important to notice that the prototype that has been developed can be used as a basis for this improvements, because its main function is to take an existing goal as input and return a list of new goals as output, about 90 percent of the code does this processing, and the basic structure of subgoals is the same as the basic structure of *or*-nodes.

Chapter 11

Conclusions and future work

In this work we have developed a narrowing calculus for unification in membership equational logic and a narrowing calculus for reachability in rewrite theories with an underlying membership equational logic, both of them making use of term typing information whenever possible. The calculi have been proved correct, and two sets of transformation rules reflecting the calculi have been developed and later implemented on a prototype.

The purpose of this work was to study the general case of conditional narrowing in membership equational logic, which has proved to be highly nondeterministic. Several calculus rules, like subject reduction, transitivity or equality, make the state space very large.

The improvements sketched in section 10.2 are an effort to keep this state space as small as possible. Implementation of this scheme was attempted at first. It was stopped recently because of lack of time for finishing it, due to its complexity, and the prototype, which is about 1.500 lines long and is still in debugging process, was written. It will be resumed in future work because, as previously said, about 90 percent of the developed code is reusable. Another improvement would be the identification of subgoals that are a renaming of another subgoal to avoid repeated computations, because fresh variables are different on each subgoal but the subgoals may represent the same unification problem.

Strategies for rule selection are a starting point for making the state space smaller, but deeper results would come from the development of membership unification algorithms or needed narrowing strategies for ACU theories.

Finally, it is worth pointing that one of the main subjects of interest for order-sorted unconditional narrowing nowadays are homomorphisms [BBBA81], especially for encryption protocol analysis [EKL+11], since current existing algorithms [AL+12] do not support neither AC properties nor order sorted theories, and it is also unknown whether there could exist a variant narrowing [ESM12] modulo homomorphism algorithm.

Bibliography

[AILS07] Luca Aceto, Anna Ing'olfsd'ottir, Kim G. Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, August 2007.

[AL+12] Siva Anantharaman, Hai Lin 0005, Christopher Lynch, Paliath Narendran, and Michaël Rusinowitch. Unification modulo homomorphic encryption. *J. Autom. Reasoning*, 48(2):135–158, 2012.

[Baa90] Franz Baader. Unification theory. In Klaus U. Schulz, editor, *IWWERT*, volume 572 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1990.

[BBBA81] S.S. Burris, S. Burris, K.P. Burris, and A.T.A. Adamson. A course in universal algebra. 1981.

[BM06] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

[BM12] Kyungmin Bae and José Meseguer. Model checking ltlr formulas under localized fairness. In Franciso Durán, editor, *WRLA*, volume 7571 of *Lecture Notes in Computer Science*, pages 99–117. Springer, 2012.

[Boc93] Alexander Bockmayr. Conditional narrowing modulo a set of equations. *Appl. Algebra Eng. Commun. Comput.*, 4:147–168, 1993.

[CDE+] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.6).

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CLD05] Hubert Comon-Lundh and Stéphanie Delaune. The finite variant property: How to get rid of some algebraic properties. In Jürgen Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

[CM96] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. *Electr. Notes Theor. Comput. Sci.*, 4:126–148, 1996.

[DLM+08] Francisco Durán, Salvador Lucas, Claude Marché, José Meseguer, and Xavier Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.

[DLM09] Francisco Durán, Salvador Lucas, and José Meseguer. Termination modulo combinations of equational theories. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCoS*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

[DM10] Francisco Durán and José Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. In Peter Csaba Ölveczky, editor, *WRLA*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010.

[DM12] Francisco Durán and José Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *J. Log. Algebr. Program.*, 81(7-8):816–850, 2012.

[DMT98] G. Denker, J. Meseguer, and C. Talcott. Protocol specification and analysis in Maude. In *In Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[Dur99] Francisco Durán. *Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[EEK+13] Serdar Erbatur, Santiago Escobar, Deepak Kapur, Zhiqiang Liu, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, Sonia Santiago, and Ralf Sasse. Asymmetric unification: A new unification paradigm for cryptographic protocol analysis. In Maria Paola Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 231–248. Springer, 2013.

[EKL+11] Santiago Escobar, Deepak Kapur, Christopher Lynch, Catherine Meadows, José Meseguer, Paliath Narendran, and Ralf Sasse. Protocol analysis in maude-npa using unification modulo homomorphic encryption.

In Peter Schneider-Kamp and Michael Hanus, editors, *PPDP*, pages 65–76. ACM, 2011.

[EMM05] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the nrl protocol analyzer: grammar generation. In Vijay Atluri, Pierangela Samarati, Ralf Küsters, and John C. Mitchell, editors, *FMSE*, pages 1–12. ACM, 2005.

[ESM12] Santiago Escobar, Ralf Sasse, and José Meseguer. Folding variant narrowing and optimal variant termination. *J. Log. Algebr. Program.*, 81(7-8):898–928, 2012.

[Fay78] M.J. Fay. *First-order Unification in an Equational Theory*. University of California, 1978.

[GKK+87] Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. An introduction to obj 3. In Stéphane Kaplan and Jean-Pierre Jouannaud, editors, *CTRS*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 1987.

[GS89] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theor. Comput. Sci.*, 67(2-3):203–260, 1989.

[HM12] Joe Hendrix and José Meseguer. Order-sorted equational unification revisited. *Electr. Notes Theor. Comput. Sci.*, 290:37–50, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[JKK83] Jean-Pierre Jouannaud, Claude Kirchner, and Hélène Kirchner. Incremental construction of unification algorithms in equational theories. In Josep Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

[KKK+96] Claude Kirchner, Hélène Kirchner, Claude Kirchner, Hélène Kirchner, Copyright C, Claude Kirchner, and Hélène Kirchner. Rewriting solving proving. Technical report, 1996.

[LMM05] Salvador Lucas, Claude Marché, and José Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, 2005.

[Mes90] José Meseguer. Rewriting as a unified model of concurrency. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer Berlin Heidelberg, 1990.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, April 1992.

[Mes97] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[MSH02] Aart Middeldorp, Taro Suzuki, and Mohamed Hamada. Complete selection functions for a lazy conditional narrowing calculus. *Journal of Functional and Logic Programming*, 2002, 2002.

[MT07] José Meseguer and Prasanna 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.

[Pet73] C. A. Petri. Concepts of net theory. In *MFCS*, pages 137–146. Mathematical Institute of the Slovak Academy of Sciences, 1973.

[Rie12] Adrián Riesco. Using narrowing to test maude specifications. In Franciso DurÃ¡n, editor, *Rewriting Logic and Its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 201–220. Springer Berlin Heidelberg, 2012.

[Tur36] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[Vir94] Patrick Viry. Rewriting: An effective model of concurrency. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer, 1994.