Rewriting Logic and Applications

Narciso Martí-Oliet

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid, Spain
narciso@esi.ucm.es

ISR 2012, Valencia, 20 July 2012

Outline

1. Introduction
    - Rewriting logic
    - Maude
2. Equational logic
    - Many-sorted equational specifications
    - Matching and rewriting
    - Order-sorted equational specifications
    - Structural axioms
    - Membership equational logic
    - Parameterization

Outline

3. Rewriting logic
   - Syntax
   - Semantics
   - System modules
   - Playing with Maude
   - Model checking
   - Reflection
   - Full Maude
   - Formal tools
   - Real-Time Maude
   - Strategy language

Outline

4. Applications
    - Overview
    - Satisfied users
    - Operational semantics
    - K framework
    - Recent features of Maude
    - Maude-NPA
    - Some work in progress

Rewriting logic

- Rewriting logic was introduced by J. Meseguer in 1990 as a unifying framework for concurrency.
- It is a simple logic to specify, reason about, and program concurrent and distributed systems.
- In the more than 20 years passed since its introduction, a large body of work by researchers around the world has contributed to the development of the logic and its applications.
- References, both to appear in Journal of Logic and Algebraic Programming, 2012,
  - José Meseguer, *Twenty years of rewriting logic*
  - Narciso Martí-Oliet, Miguel Palomino, Alberto Verdejo, *Rewriting logic bibliography by topic: 1990-2011*

Rewriting logic, Maude, and applications

- Languages whose foundations are based on rewriting logic include CafeOBJ, Elan, and Maude.
- We provide an introduction to Maude, its equational logic basis, and rewriting logic.
- We also present the rewriting logic semantics based on categories, as well as the reflective properties enjoyed by the logic.
- We mention some formal tools developed to prove properties of specifications, including a LTL model checker.
- We survey the main areas of application, including the uses of rewriting logic as a logical and semantic framework, the rewriting logic semantics program and the K framework, security applications, real-time extensions, and bioinformatics.

What is Maude?

Maude in a nutshell

http://maude.cs.uiuc.edu

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- We describe equational specification and rule-based programming in Maude, showing the difference between equations and rules.
- We use typical data structures, such as lists and binary trees, and well-known mathematical games and puzzles.
- Membership equational logic improves order-sorted algebra.
- It allows 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.

Maude in a nutshell

http://maude.cs.uiuc.edu

- Rewriting logic is a logic of concurrent change.
- It is a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency.
- It is also a good logical framework, i.e., a metalogic in which many other logics can be naturally represented and implemented.
- Moreover, rewriting logic is reflective.
- This makes possible many advanced metaprogramming and metalanguage applications.

Why declarative?

- Maude follows a long tradition of algebraic specification languages in the OBJ family, including
  - OBJ3,
  - CafeOBJ,
  - Elan.
- Computation = Deduction in an appropriate logic.
- Functional modules = (Admissible) specifications in membership equational logic.
- System modules = (Admissible) specifications in rewriting logic.
- Operational semantics based on matching and rewriting.

Many-sorted equational specifications

- Algebraic specifications are used to declare different kinds of data together with the operations that act upon them.
- It is useful to distinguish two kinds of operations:
  - constructors, used to construct or generate the data, and
  - the remaining operations, which in turn can also be classified as modifiers or observers.
- The behavior of operations is described by means of (possibly conditional) equations.
- We start with the simplest many-sorted equational specifications and incrementally add more sophisticated features.

Signatures

- The first thing a specification needs to declare are the types (or sorts) of the data being defined and the corresponding operations.
- A many-sorted signature (S, Σ) consists of
  - a sort set S, and
  - a family Σ of typed operation symbols $f : s_1 \ldots s_n \to s$.
- With the declared operations we can construct terms to denote the data being specified.
- Given a many-sorted signature (S, Σ) and an S-sorted family $X = \{X_s \mid s \in S\}$ of variables, the S-sorted set of terms is denoted

$$\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}.$$

Equations

- A Σ-equation is an expression

$$(\overline{x} : \overline{s})\ l = r$$

  where
  - $\overline{x} : \overline{s}$ is a (finite) set of variables, and
  - l and r are terms in $\mathcal{T}_{\Sigma,s}(\overline{x} : \overline{s})$ for some sort s.
- A conditional Σ-equation is an expression

$$(\overline{x} : \overline{s})\ l = r \ \text{ if } \ u_1 = v_1 \ \wedge \ldots \wedge \ u_m = v_m$$

  where $(\overline{x} : \overline{s})\ l = r$ and $(\overline{x} : \overline{s})\ u_i = v_i$ $(i = 1, \ldots, m)$ are Σ-equations.
- A many-sorted specification (S, Σ, E) consists of:
  - a signature (S, Σ), and
  - a set E of (conditional) Σ-equations.

Equational logic deduction rules

1. **Reflexivity**. $\dfrac{}{(\forall X)\, t = t}$
2. **Symmetry**. $\dfrac{(\forall X)\, t_1 = t_2}{(\forall X)\, t_2 = t_1}$
3. **Transitivity**. $\dfrac{(\forall X)\, t_1 = t_2 \quad (\forall X)\, t_2 = t_3}{(\forall X)\, t_1 = t_3}$
4. **Congruence**. For each $f \in \Sigma$, $\dfrac{(\forall X)\, t_1 = t'_1 \quad \ldots \quad (\forall X)\, t_n = t'_n}{(\forall X)\, f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}$
5. **Modus ponens**. For each sentence

$$(\overline{x} : \overline{s})\, l = r \ \text{ if } \ u_1 = v_1 \ \wedge \ldots \wedge \ u_m = v_m,$$

$$\frac{(\forall Y)\, \theta(u_1) = \theta(v_1) \quad \ldots \quad (\forall Y)\, \theta(u_m) = \theta(v_m)}{(\forall Y)\, \theta(l) = \theta(r)}$$

Semantics

- A many-sorted (S, Σ)-algebra $\mathbf{A}$ consists of:
  - a carrier set A_s for each sort $s \in S$, and
  - a function $A_f : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ for each operation symbol $f : s_1 \dots s_n \to s$.
- The meaning $[\![t]\!]_{\mathbf{A}}$ of a term t in an algebra $\mathbf{A}$ is inductively defined.
- An algebra $\mathbf{A}$ satisfies an equation $(\overline{x} : \overline{s})\ l = r$ when both terms have the same meaning: $[\![l]\!]_{\mathbf{A}} = [\![r]\!]_{\mathbf{A}}$.
- An algebra $\mathbf{A}$ satisfies a conditional equation

$$(\overline{x} : \overline{s})\ l = r \ \text{ if } \ u_1 = v_1 \ \wedge \ldots \wedge \ u_n = v_n$$

  when satisfaction of all the conditions $(\overline{x} : \overline{s})\ u_i = v_i \ (i = 1, \ldots, n)$ implies satisfaction of $(\overline{x} : \overline{s})\ l = r$.

Semantics

- The loose semantics of a many-sorted specification (S, Σ, E) is defined as the set of all (S, Σ)-algebras that satisfy all the (conditional) equations in E.
- But we are usually interested in the so-called initial semantics given by a particular algebra in this class (up to isomorphism).
- A concrete representation $\mathcal{T}_{\Sigma,E}$ of such an initial algebra is obtained by imposing a congruence relation on the term algebra $\mathcal{T}_\Sigma$ whose carrier sets are the sets of ground terms, that is, terms without variables.
- Two terms t, t' are identified by this congruence if and only if they have the same meaning in all algebras in the loose semantics (equivalently, if and only if the equation $(\forall\emptyset)\ t = t'$ can be deduced from E).

Initiality = No junk + No confusion (Burstall & Goguen)

- No junk: Every data item can be constructed using only the constants and operations in the signature.
- A data item that cannot be so constructed is junk.
- No confusion: Two data items are equivalent if and only if they can be proved equal using the equations.
- Two data items that are equivalent but cannot be proved so from the given equations are said to be confused.

Operational semantics: Matching

- Given an S-sorted family of variables X for a signature (S, Σ), a (ground) substitution is a sort-preserving map

$$\sigma : X \to \mathcal{T}_\Sigma$$

- Such a map extends uniquely to terms

$$\sigma : \mathcal{T}_\Sigma(X) \to \mathcal{T}_\Sigma$$

- Given a term $t \in \mathcal{T}_\Sigma(X)$, the pattern, and a subject ground term $u \in \mathcal{T}_\Sigma$, we say that t matches u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.

Rewriting and equational simplification

- In an admissible Σ-equation $(\overline{x} : \overline{s})\ l = r$ all variables in the righthand side r must appear among the variables of the lefthand side l.
- A term t rewrites to a term t' using such an equation if
  1. there is a subterm $t|_p$ of t at a given position p of t such that l matches $t|_p$ via a substitution σ, and
  2. $t' = t[\sigma(r)]_p$ is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$.
- We denote this step of equational simplification by $t \to_E t'$.
- It can be proved that if $t \to_E t'$ then $[\![t]\!]_{\mathbf{A}} = [\![t']\!]_{\mathbf{A}}$ for any algebra $\mathbf{A}$ satisfying E.
- We write $t \to_E^* t'$ to mean either $t = t'$ (0 steps) or $t \to_E t_1 \to_E t_2 \to_E \cdots \to_E t_n \to_E t'$ with $n \geq 0$ ($n+1$ steps).

Confluence and termination

- A set of equations E is confluent (or Church-Rosser) when any two rewritings of a term can always be joined by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.

$$
\begin{tikzcd}
 & t \arrow[dl, "E"', "*"] \arrow[dr, "*", "E"'] & \\
t_1 \arrow[dr, dotted, "E"', "*"] & & t_2 \arrow[dl, dotted, "*"', "E"] \\
 & t' &
\end{tikzcd}
$$

- A set of equations E is terminating when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \ldots$

Conditional equations

- If E is both confluent and terminating, a term t can be reduced to a unique canonical form $t{\downarrow_E}$, that is, to a term that can no longer be rewritten.
- Therefore, in order to check semantic equality of two terms $t = t'$, it is enough to check that their respective canonical forms are equal, $t{\downarrow_E} = t'{\downarrow_E}$, but, since canonical forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t{\downarrow_E} \equiv t'{\downarrow_E}$.
- This is the way to check satisfaction of equational conditions in conditional equations.
- Functional modules in Maude are assumed to be confluent and terminating, and their operational semantics is equational simplification, that is, rewriting of terms until a canonical form is obtained.

Order-sorted equational specifications

- We can often avoid some partiality by extending many-sorted equational logic to order-sorted equational logic.
- We can define subsorts corresponding to the domain of definition of a function, whenever such subsorts can be specified by means of constructors.
- Subsorts are interpreted semantically by subset inclusion.
- Operations can be overloaded.
- A term can have several different sorts. Preregularity requires each term to have a least sort that can be assigned to it.
- Maude assumes that modules are preregular, and generates warnings when a module is loaded if the property does not hold.
- Admissible equations are assumed sort-decreasing.

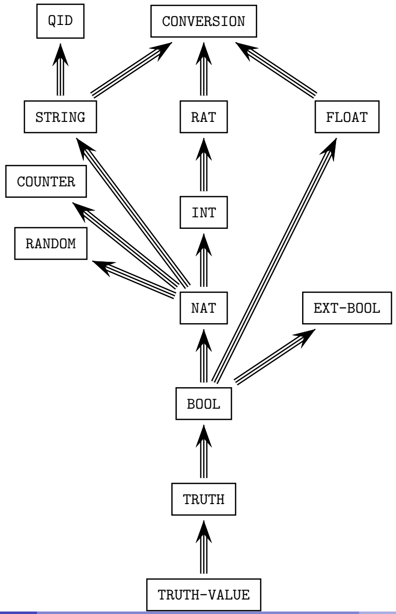
Natural numbers division

```
fmod NAT-DIV is
  sorts Nat NzNat .
  subsort NzNat < Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op _*_ : Nat Nat -> Nat .
  op _-_ : Nat Nat -> Nat .
  op _<=_ : Nat Nat -> Bool .
  op _>_ : Nat Nat -> Bool .
  op _div_ : Nat NzNat -> Nat .
  op _mod_ : Nat NzNat -> Nat .

  vars M N : Nat .
  var P : NzNat .
```

Natural numbers division

```
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq 0 * N = 0 .
  eq s(M) * N = (M * N) + N .
  eq 0 - N = 0 .
  eq s(M) - 0 = s(M) .
  eq s(M) - s(N) = M - N .
  eq 0 <= N = true .
  eq s(M) <= 0 = false .
  eq s(M) <= s(N) = M <= N .
  eq N > M = not (N <= M) .
  ceq N div P = 0 if P > N .
  ceq N div P = s((N - P) div P) if P <= N .
  ceq N mod P = N if P > N .
  ceq N mod P = (N - P) mod P if P <= N .
endfm
```

Predefined modules



Modularization

- `protecting M .`
  Importing a module M into M' in `protecting` mode intuitively means that no junk and no confusion are added to M when we include it in M'.
- `extending M .`
  The idea is to allow junk, but to rule out confusion.
- `including M .`
  No requirements are made in an `including` importation: there can now be junk and/or confusion.

Lists of natural numbers

```
fmod NAT-LIST-CONS is
  protecting NAT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .                *** empty list
  op _:_ : Nat List -> NeList [ctor] .    *** cons
  op tail : NeList -> List .
  op head : NeList -> Nat .
  op _++_ : List List -> List .           *** concatenation
  op length : List -> Nat .
  op reverse : List -> List .
  op take_from_ : Nat List -> List .
  op throw_from_ : Nat List -> List .

  vars N M : Nat .
  vars L L' : List .
```

Lists of natural numbers

```
    eq tail(N : L) = L .
    eq head(N : L) = N .
    eq [] ++ L = L .
    eq (N : L) ++ L' = N : (L ++ L') .
    eq length([]) = 0 .
    eq length(N : L) = 1 + length(L) .
    eq reverse([]) = [] .
    eq reverse(N : L) = reverse(L) ++ (N : []) .

    eq take 0 from L = [] .
    eq take N from [] = [] .
    eq take s(N) from (M : L) = M : take N from L .
    eq throw 0 from L = L .
    eq throw N from [] = [] .
    eq throw s(N) from (M : L) = throw N from L .
endfm
```

Equational attributes

- Equational attributes are a means of declaring certain kinds of structural axioms in a way that allows Maude to use these equations efficiently in a built-in way.
  - `assoc` (associativity),
  - `comm` (commutativity),
  - `idem` (idempotency),
  - `id:` t (identity, where t is the identity element),
  - left identity and right identity.
- These attributes are only allowed for binary operators satisfying some appropriate requirements depending on the attributes.

Matching and simplification modulo

- In the Maude implementation, rewriting modulo A is accomplished by using a matching modulo A algorithm.
- More precisely, given an equational theory A, a term t (corresponding to the lefthand side of an equation) and a subject term u, we say that t matches u modulo A if there is a substitution σ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and u are equal modulo the equational theory A.
- Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, Maude synthesizes a combined matching algorithm for the theory A, and does equational simplification modulo the axioms A.

A hierarchy of data types

- nonempty binary trees, with elements only in their leaves, built with a free binary constructor, that is, a constructor with no equational axioms,
- nonempty lists, built with an associative constructor,
- lists, built with an associative constructor and an identity,
- multisets (or bags), built with an associative and commutative constructor and an identity,
- sets, built with an associative, commutative, and idempotent constructor and an identity.

Basic natural numbers

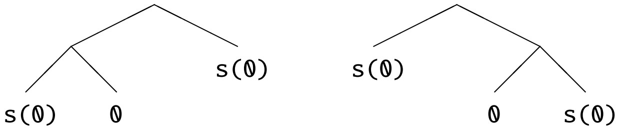
```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .

  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq max(0, M) = M .
  eq max(N, 0) = N .
  eq max(s(N), s(M)) = s(max(N, M)) .
endfm
```

Nonempty binary trees

```
fmod NAT-TREES is
  protecting BASIC-NAT .

  sorts Tree .
  subsort Nat < Tree .
  op __ : Tree Tree -> Tree [ctor] .
  op depth : Tree -> Nat .
  op width : Tree -> Nat .

  var N : Nat .
  vars T T' : Tree .

  eq depth(N) = s(0) .
  eq depth(T T') =  s(max(depth(T), depth(T'))) .
  eq width(N) = s(0) .
  eq width(T T') = width(T) + width(T') .
endfm
```

Nonempty binary trees

- An expression such as `s(0) 0 s(0)` is ambiguous because it can be parsed in two different ways, and parentheses are necessary to disambiguate `(s(0) 0) s(0)` from `s(0) (0 s(0))`.
- These two different terms correspond to the following two different trees:



Nonempty lists

```
fmod NAT-NE-LISTS is
  protecting BASIC-NAT .

  sort NeList .
  subsort Nat < NeList .
  op __ : NeList NeList -> NeList [ctor assoc] .
  op length : NeList -> Nat .
  op reverse : NeList -> NeList .

  var N : Nat .
  var L L' : NeList .

  eq length(N) = s(0) .
  eq length(L L') = length(L) + length(L') .
  eq reverse(N) = N .
  eq reverse(L L') = reverse(L') reverse(L) .
endfm
```

Lists

```
fmod NAT-LISTS is
  protecting BASIC-NAT .

  sorts NeList List .
  subsorts Nat < NeList < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
  op __ : NeList NeList -> NeList [ctor assoc id: nil] .
  op tail : NeList -> List .
  op head : NeList -> Nat .
  op length : List -> Nat .
  op reverse : List -> List .

  var N : Nat .
  var L : List .

  eq tail(N L) = L .
  eq head(N L) = N .
```

Lists

```
  eq length(nil) = 0 .
  eq length(N L) = s(0) + length(L) .
  eq reverse(nil) = nil .
  eq reverse(N L) = reverse(L) N .
endfm
```

- The alternative equation `length(L L') = length(L) + length(L')` (with `L` and `L'` variables of sort `List`) causes problems of nontermination.
- Consider the instantiation with `L'` $\mapsto$ `nil` that gives

```
length(L nil) = length(L) + length(nil)
              = length(L nil) + length(nil)
              = (length(L) + length(nil)) + length(nil)
              = ...
```

because of the identification `L = L nil`.

Multisets

```
fmod NAT-MSETS is
  protecting BASIC-NAT .
  sort Mset .
  subsorts Nat < Mset .
  op empty-mset : -> Mset [ctor] .
  op __ : Mset Mset -> Mset [ctor assoc comm id: empty-mset] .
  op size : Mset -> Nat .
  op mult : Nat Mset -> Nat .
  op _in_ : Nat Mset -> Bool .

  vars N N' : Nat .
  var  S : Mset .

  eq size(empty-mset) = 0 .
  eq size(N S) = s(0) + size(S) .
  eq mult(N, empty-mset) = 0 .
  eq mult(N, N S) = s(0) + mult(N, S) .
  ceq mult(N, N' S) = mult(N, S) if N =/= N' .
  eq N in S = (mult(N, S) =/= 0) .
endfm
```

Sets

```
fmod NAT-SETS is
  protecting BASIC-NAT .
  sort Set .
  subsorts Nat < Set .
  op empty-set : -> Set [ctor] .
  op __ : Set Set -> Set [ctor assoc comm id: empty-set] .

  vars N N' : Nat .
  vars S S' : Set .

  eq N N = N .
```

The idempotency equation is stated only for singleton sets, because stating it for arbitrary sets in the form `S S = S` would cause nontermination due to the identity attribute:

$$\texttt{empty-set} = \texttt{empty-set empty-set} \to \texttt{empty-set} \ldots$$

Sets

```
  op _in_ : Nat Set -> Bool .
  op delete : Nat Set -> Set .
  op card : Set -> Nat .

  eq N in empty-set = false .
  eq N in (N' S) = (N == N') or (N in S) .
  eq delete(N, empty-set) = empty-set .
  eq delete(N, N S) = delete(N, S) .
  ceq delete(N, N' S) = N' delete(N, S) if N =/= N' .
  eq card(empty-set) = 0 .
  eq card(N S) = s(0) + card(delete(N,S)) .
endfm
```

The equations for `delete` and `card` make sure that further occurrences of `N` in `S` on the righthand side are also deleted or not counted, resp., because we cannot rely on the order in which equations are applied.

Membership equational logic specifications

- In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of sorted lists, for example, defined by a property over lists.
- There is also a different problem of a more syntactic character. For example, with operations of difference and division on natural numbers, the term `s(s(s(0))) div (s(s(0)) - s(0))` would not be well formed, because the subterm `s(s(0)) - s(0)` has least sort `Nat`, while the `div` operation would expect its second argument to be of sort `NzNat` < `Nat`.
- This is too restrictive and makes most (really) order-sorted specifications useless, unless there is a mechanism that gives at parsing time the benefit of the doubt to this kind of terms.
- Membership equational logic solves both problems, by introducing sorts as predicates and allowing subsort definition by means of conditions involving equations and/or sort predicates.

Membership equational logic

- A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ where K is a set of kinds, (K, Σ) is a many-kinded signature, and $S = \{S_k\}_{k \in K}$ is a K-kinded set of sorts.
- An Ω-algebra is then a (K, Σ)-algebra $\mathbf{A}$ together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$.
- Atomic formulas are either Σ-equations, or membership assertions of the form $t : s$, where the term t has kind k and $s \in S_k$.
- General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables.

$$(\forall X)\ \ l = r \ \text{ if } \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

$$(\forall X)\ \ t : s \ \text{ if } \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

Membership equational logic deduction rules

1. **Reflexivity.** $\dfrac{}{(\forall X)\ t = t}$
2. **Symmetry.** $\dfrac{(\forall X)\ t_1 = t_2}{(\forall X)\ t_2 = t_1}$
3. **Transitivity.** $\dfrac{(\forall X)\ t_1 = t_2 \quad (\forall X)\ t_2 = t_3}{(\forall X)\ t_1 = t_3}$
4. **Congruence.** For each $f \in \Sigma$, $\dfrac{(\forall X)\ t_1 = t_1' \quad \ldots \quad (\forall X)\ t_n = t_n'}{(\forall X)\ f(t_1, \ldots, t_n) = f(t_1', \ldots, t_n')}$
5. **Membership.** $\dfrac{(\forall X)\ t_1 = t_2 \quad (\forall X)\ t_1 : s}{(\forall X)\ t_2 : s}$

Membership equational logic deduction rules

6. **Modus ponens 1**. For each sentence

$$(\forall X)\ \ l = r\ \text{ if }\ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j),$$

$$\frac{(\forall Y)\,\theta(u_i) = \theta(v_i)\ \text{for all } i, \quad (\forall Y)\,\theta(w_j) : s_j\ \text{for all } j}{(\forall Y)\,\theta(l) = \theta(r)}$$

7. **Modus ponens 2**. For each sentence

$$(\forall X)\ \ t : s\ \text{ if }\ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j),$$

$$\frac{(\forall Y)\,\theta(u_i) = \theta(v_i)\ \text{for all } i, \quad (\forall Y)\,\theta(w_j) : s_j\ \text{for all } j}{(\forall Y)\,\theta(t) : s}$$

Membership equational logic in Maude

- Maude functional modules are membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.
- Maude does automatic kind inference from the sorts declared by the user and their subsort relations.
- Kinds are not declared explicitly, and correspond to the connected components of the subsort relation.
- The kind corresponding to a sort `s` is denoted `[s]`.
- If `NzNat` < `Nat`, then `[NzNat]` = `[Nat]`.

Membership equational logic in Maude

- An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

  can be understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

  together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

- A subsort declaration `NzNat < Nat` can be understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

Sorted lists

```
fmod NAT-SORTED-LIST is
  protecting NAT-LIST-CONS .

  sorts SortedList NeSortedList .
  subsort NeSortedList < SortedList NeList < List .

  op insertion-sort : List -> SortedList .
  op insert-list : SortedList Nat -> SortedList .

  op mergesort : List -> SortedList .
  op merge : SortedList SortedList -> SortedList [comm] .

  op quicksort : List -> SortedList .
  op leq-elems : List Nat -> List .
  op gr-elems : List Nat -> List .

  vars N M : Nat .
  vars L L' : List .
  vars OL OL' : SortedList .
  var NEOL : NeSortedList .
```

Sorted lists

```
mb [] : SortedList .
mb N : [] : NeSortedList .
cmb N : NEOL : NeSortedList if N <= head(NEOL) .

eq insertion-sort([]) = [] .
eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .

eq insert-list([], M) = M : [] .
ceq insert-list(N : OL, M) = M : N : OL if M <= N .
ceq insert-list(N : OL, M) = N : insert-list(OL, M) if M > N .

eq mergesort([]) = [] .
eq mergesort(N : []) = N : [] .
ceq mergesort(L) =
    merge(mergesort(take (length(L) quo 2) from L),
          mergesort(throw (length(L) quo 2) from L))
 if length(L) > s(0) .
```

Sorted lists

```
  eq merge(OL, []) = OL .
  ceq merge(N : OL, M : OL') = N : merge(OL, M : OL') if N <= M .

  eq quicksort([]) = [] .
  eq quicksort(N : L)
    = quicksort(leq-elems(L,N)) ++ (N : quicksort(gr-elems(L,N))) .

  eq leq-elems([], M) = [] .
  ceq leq-elems(N : L, M) = N : leq-elems(L, M) if N <= M .
  ceq leq-elems(N : L, M) = leq-elems(L, M) if N > M .
  eq gr-elems([], M) = [] .
  ceq gr-elems(N : L, M) = gr-elems(L, M) if N <= M .
  ceq gr-elems(N : L, M) = N : gr-elems(L, M) if N > M .
endfm
```

Parameterization: theories

- Parameterized datatypes use theories to specify the requirements that the parameter must satisfy.
- A (functional) theory is a membership equational specification whose semantics is loose.
- Equations in a theory are not used for rewriting or equational simplication and, thus, they need not be confluent or terminating.
- Simplest theory only requires existence of a sort:

```
fth TRIV is
  sort Elt .
endfth
```

Order theories

- Theory requiring a strict total order over a given sort:

```
fth STOSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

Order theories

- Theory requiring a nonstrict total order over a given sort:

```
fth TOSET is
  including STOSET .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= X = true [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

Parameterization: views

- Theories are used in a parameterized module expression such as

  ```
  fmod LIST{X :: TRIV} is  ...  endfm
  ```

  to make explicit the requirements over the argument module.
- A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module.
- Each view declaration has an associated set of proof obligations, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. This may in general require inductive proof techniques.
- In many simple cases it is completely obvious:

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

Parameterization: instantiation

- A module expression such as LIST{Nat} denotes the instantiation of the parameterized module LIST{X :: TRIV} by means of the previous view Nat.

$$\begin{tikzcd}
\texttt{TRIV} \arrow[r, "\texttt{Nat}"] \arrow[d] & \texttt{NAT} \arrow[d] \\
\texttt{LIST\{X :: TRIV\}} \arrow[r] & \texttt{LIST\{Nat\}}
\end{tikzcd}$$

- Views can also go from theories to theories, meaning an instantiation that is still parameterized.

```
view Toset from TRIV to TOSET is
  sort Elt to Elt .
endv
```

- It is possible to have more than one view from a theory to a module or to another theory.

Parameterized lists

```
fmod LIST-CONS{X :: TRIV} is
  protecting NAT .

  sorts NeList{X} List{X} .
  subsort NeList{X} < List{X} .

  op [] : -> List{X} [ctor] .
  op _:_ : X$Elt List{X} -> NeList{X} [ctor] .
  op tail : NeList{X} -> List{X} .
  op head : NeList{X} -> X$Elt .

  var E : X$Elt .
  var N : Nat .
  vars L L' : List{X} .

  eq tail(E : L) = L .
  eq head(E : L) = E .
```

Parameterized lists

```
  op _++_ : List{X} List{X} -> List{X} .
  op length : List{X} -> Nat .
  op reverse : List{X} -> List{X} .
  op take_from_ : Nat List{X} -> List{X} .
  op throw_from_ : Nat List{X} -> List{X} .

  eq [] ++ L = L .
  eq (E : L) ++ L' = E : (L ++ L') .
  eq length([]) = 0 .
  eq length(E : L) = 1 + length(L) .
  eq reverse([]) = [] .
  eq reverse(E : L) = reverse(L) ++ (E : []) .
  eq take 0 from L = [] .
  eq take N from [] = [] .
  eq take s(N) from (E : L) = E : take N from L .
  eq throw 0 from L = L .
  eq throw N from [] = [] .
  eq throw s(N) from (E : L) = throw N from L .
endfm
```

Parameterized sorted lists

```
view Toset from TRIV to TOSET is
  sort Elt to Elt .
endv

fmod SORTED-LIST{X :: TOSET} is
  protecting LIST-CONS{Toset}{X} .

  sorts SortedList{X} NeSortedList{X} .
  subsorts NeSortedList{X} < SortedList{X} < List{Toset}{X} .
  subsort NeSortedList{X} < NeList{Toset}{X} .

  vars N M : X$Elt .
  vars L L' : List{Toset}{X} .
  vars OL OL' : SortedList{X} .
  var NEOL : NeSortedList{X} .
```

Parameterized sorted lists

```
    mb [] : SortedList{X} .
    mb (N : []) : NeSortedList{X} .
    cmb (N : NEOL) : NeSortedList{X} if N <= head(NEOL) .

    op insertion-sort : List{Toset}{X} -> SortedList{X} .
    op insert-list : SortedList{X} X$Elt -> SortedList{X} .

    op mergesort : List{Toset}{X} -> SortedList{X} .
    op merge : SortedList{X} SortedList{X} -> SortedList{X} [comm] .

    op quicksort : List{Toset}{X} -> SortedList{X} .
    op leq-elems : List{Toset}{X} X$Elt -> List{Toset}{X} .
    op gr-elems : List{Toset}{X} X$Elt -> List{Toset}{X} .

    *** equations as before
endfm
```

Parameterized sorted lists

```
view NatAsToset from TOSET to NAT is
  sort Elt to Nat .
endv

fmod SORTED-LIST-TEST is
  protecting SORTED-LIST{NatAsToset} .
endfm

Maude> red insertion-sort(5 : 4 : 3 : 2 : 1 : 0 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red mergesort(5 : 3 : 1 : 0 : 2 : 4 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red quicksort(0 : 1 : 2 : 5 : 4 : 3 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []
```

Binary trees

```
fmod BIN-TREE{X :: TRIV} is
  protecting LIST-CONS{X} .

  sorts NeBinTree{X} BinTree{X} .
  subsort NeBinTree{X} < BinTree{X} .

  op empty : -> BinTree{X} [ctor] .
  op _[_]_ : BinTree{X} X$Elt BinTree{X} -> NeBinTree{X} [ctor] .
  ops left right : NeBinTree{X} -> BinTree{X} .
  op root : NeBinTree{X} -> X$Elt .

  var E : X$Elt .
  vars L R : BinTree{X} .
  vars NEL NER : NeBinTree{X} .
  ..............

 endfm
```

Binary search trees

```
fmod SEARCH-TREE{X :: STOSET, Y :: CONTENTS} is
  ..............

  mb empty : SearchTree{X, Y} .
  mb empty [SRec] empty : NeSearchTree{X, Y} .
  cmb L' [SRec] empty : NeSearchTree{X, Y}
    if key(max(L')) < key(SRec) .
  cmb empty [SRec] R' : NeSearchTree{X, Y}
    if key(SRec) < key(min(R')) .
  cmb L' [SRec] R' : NeSearchTree{X, Y}
    if key(max(L')) < key(SRec) /\ key(SRec) < key(min(R')) .

  ..............
```

Rewriting logic

- We arrive at the main idea behind rewriting logic by dropping symmetry and the equational interpretation of rules.
- We interpret a rule $t \rightarrow t'$ computationally as a local concurrent transition of a system, and logically as an inference step from formulas of type t to formulas of type t'.
- Rewriting logic is a logic of becoming or change, that allows us to specify the dynamic aspects of systems.
- Representation of systems in rewriting logic:
  - The static part is specified as an equational theory.
  - The dynamics is specified by means of possibly conditional rules that rewrite terms, representing parts of the system, into others.
  - The rules need only specify the part of the system that actually changes: the frame problem is avoided.

Rewriting logic

- A rewriting logic signature is an equational specification (Ω, E) that makes explicit the set of equations in order to emphasize that rewriting will operate on congruence classes of terms modulo E.
- Sentences are rewrites of the form $[t]_E \longrightarrow [t']_E$.
- A rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ consists of:
  - a signature (Ω, E),
  - a set L of labels, and
  - a set R of labelled rewrite rules $\quad r : [t]_E \longrightarrow [t']_E$ where r is a label and $[t]_E, [t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$.
- The most general form of a rewrite rule is conditional:

$$r : t \to t' \;\; \mathit{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

The meaning of rewriting logic

$$\begin{array}{lclcl}
\textit{State} & \longleftrightarrow & \textit{Term} & \longleftrightarrow & \textit{Proposition} \\
\textit{Transition} & \longleftrightarrow & \textit{Rewriting} & \longleftrightarrow & \textit{Deduction} \\
\textit{Distributed} & \longleftrightarrow & \textit{Algebraic} & \longleftrightarrow & \textit{Propositional} \\
\quad\textit{structure} & & \quad\textit{structure} & & \quad\textit{structure}
\end{array}$$

Rewriting logic deduction rules (unconditional, unsorted)

1. **Reflexivity**. $\dfrac{}{[t] \longrightarrow [t]}$
2. **Transitivity**. $\dfrac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$
3. **Congruence**. For each $f \in \Sigma$, $\dfrac{[t_1] \longrightarrow [t'_1] \quad \dots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \dots, t_n)] \longrightarrow [f(t'_1, \dots, t'_n)]}$
4. **Replacement**. For each rewrite rule

$$r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)],$$

$$\frac{[w_1] \longrightarrow [w'_1] \quad \dots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

Rewriting logic semantics

- A rewrite theory is a static description of what a system can do. The meaning should be given by a model of its actual behavior.
- A rewrite theory $\mathcal{R} = (\Omega, E, L, R)$ has initial and free models that capture computationally the intuitive idea of a "rewrite system" whose states are E-equivalence classes of terms and whose transitions are concurrent rewritings using the rules in R.
- Logically, we can view such models as "logical systems" in which formulas are validly rewritten to other formulas by concurrent rewritings which correspond to proofs.
- The free model is a category $\mathcal{T}_{\mathcal{R}}(X)$ whose objects are equivalence classes of terms $[t] \in T_{\Omega,E}(X)$ and whose morphisms are equivalence classes of "proof terms" representing proofs in rewriting deduction, i.e., concurrent $\mathcal{R}$-rewrites.

Rewriting logic semantics: proof term generation

1. **Identities**. $\dfrac{}{[t] : [t] \longrightarrow [t]}$
2. **Composition**. $\dfrac{\alpha : [t_1] \longrightarrow [t_2] \qquad \beta : [t_2] \longrightarrow [t_3]}{\alpha;\beta : [t_1] \longrightarrow [t_3]}$
3. **Σ-structure**. For each $f \in \Sigma$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t_1'] \quad \dots \quad \alpha_n : [t_n] \longrightarrow [t_n']}{f(\alpha_1, \dots, \alpha_n) : [f(t_1, \dots, t_n)] \longrightarrow [f(t_1', \dots, t_n')]}$$

4. **Replacement**. For each rewrite rule

$$r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)],$$

$$\frac{\alpha_1 : [w_1] \longrightarrow [w_1'] \quad \dots \quad \alpha_n : [w_n] \longrightarrow [w_n']}{r(\alpha_1, \dots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

Rewriting logic semantics: proof term equalities

1. **Category**.
   - *Associativity*. For all α, β, γ

$$(\alpha;\beta);\gamma = \alpha;(\beta;\gamma)$$

   - *Identities*. For each $\alpha : [t] \longrightarrow [t']$

$$\alpha;[t'] = \alpha \qquad\qquad [t];\alpha = \alpha$$

2. **Functoriality of the Σ-algebraic structure**. For each $f \in \Sigma$,
   - *Preservation of composition*. For all α_i, β_i,

$$f(\alpha_1;\beta_1, \ldots, \alpha_n;\beta_n) = f(\alpha_1, \ldots, \alpha_n); f(\beta_1, \ldots, \beta_n)$$

   - *Preservation of identities*.

$$f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$$

Rewriting logic semantics: proof term equalities

3. **Axioms in** E.
   For each equation $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E, for all α_i,

   $$t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$$

4. **Exchange**.
   For each $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

   $$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{r(\overline{\alpha}) = r(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); r(\overline{[w']})}$$

$\mathcal{R}$-Systems

- The category $\mathcal{T}_{\mathcal{R}}(X)$ is one among many models that can be assigned to the rewrite theory $\mathcal{R}$.
- Given a rewrite theory $\mathcal{R} = (\Omega, E, L, R)$, an $\mathcal{R}$-system $\mathcal{S}$ is a category $\mathcal{S}$ together with:
  - a (Σ, E)-algebra structure given by a family of functors

    $$\{f_{\mathcal{S}} : \mathcal{S}^n \longrightarrow \mathcal{S} \mid f \in \Sigma_n\}$$

    satisfying E, i.e., for any equation $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ in E we have an identity of functors $t_{\mathcal{S}} = t'_{\mathcal{S}}$, where the functor $t_{\mathcal{S}}$ is defined inductively from the functors $f_{\mathcal{S}}$.
  - for each rewrite rule $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$ in R a natural transformation $r_{\mathcal{S}} : t_{\mathcal{S}} \Rightarrow t'_{\mathcal{S}}$.

A more detailed deduction system

R. Bruni, J. Meseguer / Theoretical Computer Science 360 (2006) 386–414

$$\frac{t \in \mathbb{T}_\Sigma(X)_k}{(\forall X)\, t \to t}\ \textbf{Reflexivity} \qquad \frac{(\forall X)\, t_1 \to t_2, \qquad (\forall X)\, t_2 \to t_3}{(\forall X)\, t_1 \to t_3}\ \textbf{Transitivity}$$

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

$$\frac{\begin{array}{c} f \in \Sigma_{k_1 \cdots k_n, k}, \qquad t_i, t'_i \in \mathbb{T}_\Sigma(X)_{k_i} \text{ for } i \in [1,n] \\ t'_i = t_i \text{ for } i \in \phi(f), \qquad (\forall X)\, t_j \to t'_j \text{ for } j \in \nu(f) \end{array}}{(\forall X)\, f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n)}\ \textbf{Congruence}$$

$$\frac{\begin{array}{c} (\forall X)\, r{:}t \to t' \text{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \to t'_l \in R \\ \theta, \theta'{:}X \to \mathbb{T}_\Sigma(Y), \qquad \theta(x) = \theta'(x) \text{ for } x \in \phi(t,t') \\ E \vdash (\forall Y)\, \theta(p_i) = \theta(q_i) \text{ for } i \in I, \qquad E \vdash (\forall Y)\, \theta(w_j) : s_j \text{ for } j \in J \\ (\forall Y)\, \theta(t_l) \to \theta(t'_l) \text{ for } l \in L, \qquad (\forall Y)\, \theta(x) \to \theta'(x) \text{ for } x \in \nu(t,t') \end{array}}{(\forall Y)\, \theta(t) \to \theta'(t')}\ \begin{array}{l}\textbf{Nested}\\ \textbf{Replacement}\end{array}$$

Fig. 5. Deduction rules for generalized rewrite theories.

System modules

- System modules in Maude correspond to rewrite theories in rewriting logic.
- A rewrite theory has both rules and equations, so that rewriting is performed modulo such equations.
- The equations are divided into
  - a set A of structural axioms, for which matching algorithms exist in Maude, and
  - a set E of equations that are Church-Rosser and terminating modulo A;

  that is, the equational part must be equivalent to a functional module.

System modules

- The rules R in the module must be coherent with the equations E modulo A, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken.

$$\begin{tikzcd}
t \arrow[rr, "1"', "R/A"] \arrow[dd, "!"', "E/A"] & & t' \arrow[dr, dotted, "!", "E/A"'] & \\
 & & & w \\
u \arrow[rr, dotted, "1", "R/A"'] & & u' \arrow[ur, dotted, "!", "E/A"'] &
\end{tikzcd}$$

- A simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.
- In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A.

Hopping rabbits

- Initial configuration (for 3 rabbits in each team):

- Final configuration:

- X-rabbits move to the right.
- O-rabbits move to the left.
- A rabbit is allowed to advance one position if that position is empty.
- A rabbit can jump over a rival if the position behind it is free.

Hopping rabbits

```
mod RABBIT-HOP is

  ***  each rabbit is represented as a constant
  ***  a special rabbit for the empty position

  sort Rabbit .
  ops x o free : -> Rabbit .

  *** a game state is represented
  *** as a nonempty list of rabbits

  sort RabbitList .
  subsort Rabbit < RabbitList .
  op __ : RabbitList RabbitList -> RabbitList [assoc] .
```

Hopping rabbits

```
  *** rules (transitions) for game moves

  rl [xAdvances] : x free => free x .
  rl [xJumps] : x o free => free o x .
  rl [oAdvances] : free o => o free .
  rl [oJumps] : free x o => o x free .

  *** auxiliary operation to build initial states

  protecting NAT .

  op initial : Nat -> RabbitList .
  var  N : Nat .
  eq initial(0) = free .
  eq initial(s(N)) = x initial(N) o .
endm
```

Hopping rabbits

```
Maude> search initial(3) =>* o o o free x x x .

Solution 1 (state 71)
empty substitution

No more solutions.

Maude> show path labels 71 .
xAdvances     oJumps     oAdvances
xJumps        xJumps     xAdvances
oJumps        oJumps     oJumps
xAdvances     xJumps     xJumps
oAdvances     oJumps     xAdvances

Maude> show path 71 .
state 0, RabbitList: x x x free o o o
===[ rl x free => free x [label xAdvances] . ]===>
state 1, RabbitList: x x free x o o o
...
```

The three basins puzzle

- We have three basins with capacities of 3, 5, and 8 gallons.
- There is an unlimited supply of water.
- The goal is to get 4 gallons in any of the basins.
- Practical application: in the movie *Die Hard: With a Vengeance*, McClane and Zeus have to deactivate a bomb with this system.
- A basin is represented with the constructor `basin`, having two natural numbers as arguments: the first one is the basin capacity and the second one is how much it is filled.
- We can think of a basin as an object with two attributes.
- This leads to an object-based style of programming, where objects change their attributes as result of interacting with other objects.
- Interactions are represented as rules on configurations that are nonempty multisets of objects.

The three basins puzzle

```
mod DIE-HARD is
  protecting NAT .

  *** objects
  sort Basin .
  op basin : Nat Nat -> Basin .    *** capacity / content

  *** configurations / multisets of objects
  sort BasinSet .
  subsort Basin < BasinSet .
  op __ : BasinSet BasinSet -> BasinSet [assoc comm] .

  *** auxiliary operation to represent initial state
  op initial : -> BasinSet .
  eq initial = basin(3, 0) basin(5, 0) basin(8,0) .
```

The three basins puzzle

```
  *** possible moves as four rules
  vars M1 N1 M2 N2 : Nat .

  rl [empty] : basin(M1, N1) => basin(M1, 0) .

  rl [fill] : basin(M1, N1) => basin(M1, M1) .

  crl [transfer1] : basin(M1, N1) basin(M2, N2)
    => basin(M1, 0) basin(M2, N1 + N2)
   if N1 + N2 <= M2 .

  crl [transfer2] : basin(M1, N1) basin(M2, N2)
    => basin(M1, sd(N1 + N2, M2)) basin(M2, M2)
   if N1 + N2 > M2 .

   *** sd is symmetric difference in predefined NAT
endm
```

The three basins puzzle

```
Maude> search [1] initial =>* basin(N:Nat, 4) B:BasinSet .

Solution 1 (state 75)
B:BasinSet --> basin(3, 3) basin(8, 3)
N:Nat --> 5

Maude> show path 75 .
state 0, BasinSet: basin(3, 0) basin(5, 0) basin(8, 0)
===[ rl ... fill ]===>
state 2, BasinSet: basin(3, 0) basin(5, 5) basin(8, 0)
===[ crl ... transfer2 ]===>
state 9, BasinSet: basin(3, 3) basin(5, 2) basin(8, 0)
===[ crl ... transfer1 ]===>
state 20, BasinSet: basin(3, 0) basin(5, 2) basin(8, 3)
===[ crl ... transfer1 ]===>
state 37, BasinSet: basin(3, 2) basin(5, 0) basin(8, 3)
===[ rl ... fill ]===>
state 55, BasinSet: basin(3, 2) basin(5, 5) basin(8, 3)
===[ crl ... transfer2 ]===>
state 75, BasinSet: basin(3, 3) basin(5, 4) basin(8, 3)
```

Crossing the bridge

- The four components of U2 are in a tight situation. Their concert starts in 17 minutes and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time.
- It is already dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a flashlight. Unfortunately, they only have one.
- Knowing that Bono, Edge, Adam, and Larry take 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

Crossing the bridge

- The current state of the group can be represented by a multiset (a term of sort `Group` below) consisting of performers, the flashlight, and a watch to keep record of the time.
- The flashlight and the performers have a `Place` associated to them, indicating whether their current position is to the left or to the right of the bridge.
- Each performer, in addition, also carries the time it takes him to cross the bridge.
- In order to change the position from `left` to `right` and vice versa, we use an auxiliary operation `changePos`.
- The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second one for when there are two.

Crossing the bridge

```
mod U2 is
  protecting NAT .

  sorts Performer Object Group Place .
  subsorts Performer Object < Group .

  ops left right : -> Place .
  op flashlight : Place -> Object .
  op watch : Nat -> Object .
  op performer : Nat Place -> Performer .
  op __ : Group Group -> Group [assoc comm] .

  op changePos : Place -> Place .

  eq changePos(left) = right .
  eq changePos(right) = left .
```

Crossing the bridge

```
  op initial : -> Group .
  eq initial
    = watch(0) flashlight(left) performer(1, left)
      performer(2, left) performer(5, left) performer(10, left) .

  var  P : Place .
  vars M N N1 N2 : Nat .

  rl [one-crosses] :
    watch(M) flashlight(P) performer(N, P)
    => watch(M + N) flashlight(changePos(P))
       performer(N, changePos(P)) .

  crl [two-cross] :
    watch(M) flashlight(P) performer(N1, P) performer(N2, P)
    => watch(M + N1) flashlight(changePos(P))
       performer(N1, changePos(P))
       performer(N2, changePos(P))
    if N1 > N2 .
endm
```

Crossing the bridge

- A solution can be found by looking for a state in which all performers and the flashlight are to the `right` of the bridge.
- The search command is invoked with a such that clause that allows to introduce a condition that solutions have to fulfill, in our example, that the total time is less than or equal to 17 minutes:

```
Maude> search [1] initial
         =>* flashlight(right) watch(N:Nat)
             performer(1, right) performer(2, right)
             performer(5, right) performer(10, right)
         such that N:Nat <= 17 .

Solution 1 (state 402)
N --> 17
```

Crossing the bridge

- The solution takes exactly 17 minutes (a happy ending after all!) and the complete sequence of appropriate actions can be shown with the command

```
Maude> show path 402 .
```

- After sorting out the information, it becomes clear that Bono and Edge have to be the first to cross. Then Bono returns with the flashlight, which gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.
- Note that, in order for the `search` command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, increasingly taking a larger amount of time, and it will never end.

Model checking

- Two levels of specification:
  - a system specification level, provided by the rewrite theory specified by that system module, and
  - a property specification level, given by some properties that we want to state and prove about our module.
- Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens), related to the infinite behavior of a system.
- Maude 2 includes a model checker to prove properties expressed in linear temporal logic (LTL).

Linear temporal logic

- Main connectives:
  - True: $\top \in \mathrm{LTL}(AP)$.
  - Atomic propositions: If $p \in AP$, then $p \in \mathrm{LTL}(AP)$.
  - Next operator: If $\varphi \in \mathrm{LTL}(AP)$, then $\bigcirc\varphi \in \mathrm{LTL}(AP)$.
  - Until operator: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then $\varphi \, \mathcal{U} \, \psi \in \mathrm{LTL}(AP)$.
  - Boolean connectives: If $\varphi, \psi \in \mathrm{LTL}(AP)$, then the formulae $\neg\varphi$, and $\varphi \vee \psi$ are in $\mathrm{LTL}(AP)$.
- Other Boolean connectives:
  - False: $\bot = \neg\top$
  - Conjunction: $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
  - Implication: $\varphi \to \psi = (\neg\varphi) \vee \psi$.

Linear temporal logic

- Other temporal operators:
  - Eventually: $\Diamond\varphi = \top\ \mathcal{U}\ \varphi$
  - Henceforth: $\Box\varphi = \neg\Diamond\neg\varphi$
  - Release: $\varphi\ \mathcal{R}\ \psi = \neg((\neg\varphi)\ \mathcal{U}\ (\neg\psi))$
  - Unless: $\varphi\ \mathcal{W}\ \psi = (\varphi\ \mathcal{U}\ \psi) \vee (\Box\varphi)$
  - Leads-to: $\varphi \rightsquigarrow \psi = \Box(\varphi \to (\Diamond\psi))$
  - Strong implication: $\varphi \Rightarrow \psi = \Box(\varphi \to \psi)$
  - Strong equivalence: $\varphi \Leftrightarrow \psi = \Box(\varphi \leftrightarrow \psi)$.

Linear temporal logic

- Before considering their formal meaning, let us note that the intuition behind the main temporal connectives is the following:
  - $\top$ is a formula that always holds at the current state.
  - $\bigcirc\varphi$ holds at the current state if φ holds at the state that follows.
  - $\varphi\,\mathcal{U}\,\psi$ holds at the current state if ψ is eventually satisfied at a future state and, until that moment, φ holds at all intermediate states.
  - $\Box\varphi$ holds if φ holds at every state from now on.
  - $\Diamond\varphi$ holds if φ holds at some state in the future.

Kripke structures

- A Kripke structure is a triple $\mathcal{A} = (A, \to_{\mathcal{A}}, L)$ such that
  - A is a set, called the set of states,
  - $\to_{\mathcal{A}}$ is a total binary relation on A, called the transition relation, and
  - $L : A \longrightarrow \mathcal{P}(AP)$ is a function, called the labeling function, associating to each state $a \in A$ the set $L(a)$ of those atomic propositions in AP that hold in the state a.
- A path in a Kripke structure $\mathcal{A}$ is a function $\pi : \mathbb{N} \longrightarrow A$ with $\pi(i) \to_{\mathcal{A}} \pi(i+1)$ for every i.
- We use π^i to refer to the suffix of π starting at $\pi(i)$.

Kripke structures: semantics

- The semantics of LTL is defined by means of a satisfaction relation between a Kripke structure $\mathcal{A}$, a state $a \in A$, and a formula φ:

$$\mathcal{A}, a \models \varphi \iff \mathcal{A}, \pi \models \varphi \quad \text{for all paths } \pi \text{ with } \pi(0) = a.$$

- The satisfaction relation for paths $\mathcal{A}, \pi \models \varphi$ is defined by structural induction on φ:

$$\begin{array}{lcl}
\mathcal{A}, \pi \models p & \iff & p \in L(\pi(0)) \\
\mathcal{A}, \pi \models \top & \iff & \mathit{true} \\
\mathcal{A}, \pi \models \varphi \vee \psi & \iff & \mathcal{A}, \pi \models \varphi \text{ or } \mathcal{A}, \pi \models \psi \\
\mathcal{A}, \pi \models \neg\varphi & \iff & \mathcal{A}, \pi \not\models \varphi \\
\mathcal{A}, \pi \models \bigcirc\varphi & \iff & \mathcal{A}, \pi^1 \models \varphi \\
\mathcal{A}, \pi \models \varphi \,\mathcal{U}\, \psi & \iff & \text{there exists } n \in \mathbb{N} \text{ such that } \mathcal{A}, \pi^n \models \psi \\
 & & \text{and, for all } m < n,\ \mathcal{A}, \pi^m \models \varphi
\end{array}$$

The semantics of the remaining Boolean and temporal operators (e.g., $\bot$, $\wedge$, $\rightarrow$, $\Box$, $\Diamond$, $\mathcal{R}$, and $\rightsquigarrow$) can be derived from these.

Kripke structures associated to rewrite theories

- Given a system module M specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, we
  - choose a kind k in M as our kind of states;
  - define some state predicates Π and their semantics in a module, say M-PREDS, protecting M by means of the operation

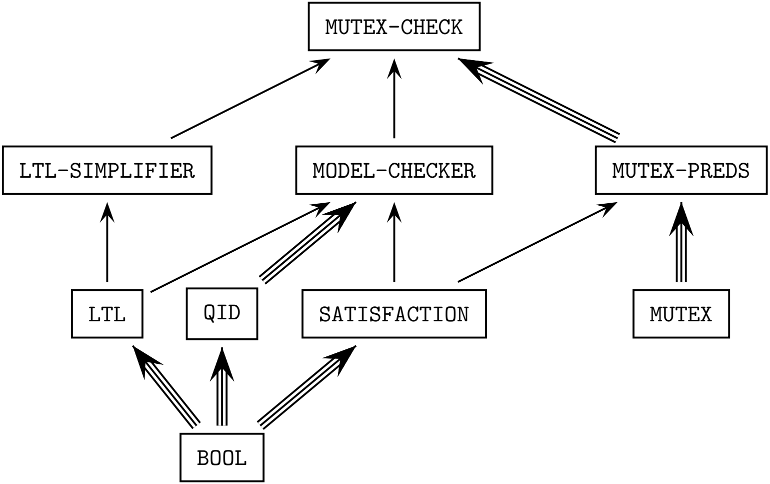
    ```
    op _|=_ : State Prop -> Bool .
    ```

    coming from the predefined SATISFACTION module.
- Then we get a Kripke structure

$$\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\to^1_{\mathcal{R}})^\bullet, L_\Pi).$$

- Under some assumptions on M and M-PREDS, including that the set of states reachable from $[t]$ is finite, the relation $\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$ becomes decidable.

Model-checking modules



Mutual exclusion: processes

```
mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: none] .

  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
  ops * $ : -> Token [ctor] .

  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
endm
```

Mutual exclusion: basic properties

```
mod MUTEX-PREDS is
  protecting MUTEX .
  including SATISFACTION .
  subsort Conf < State .

  op crit : Name -> Prop .
  op wait : Name -> Prop .

  var N : Name .
  var C : Conf .
  var P : Prop .

  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
  eq C |= P = false [owise] .
endm
```

Model checking mutual exclusion

```
mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a, wait] [b, wait] .
  eq initial2 = * [a, wait] [b, wait] .
endm

Maude> red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true
```

A strong liveness property

If a process waits infinitely often, then it is in its critical section infinitely often.

```
Maude> red modelCheck(initial1, ([]<> wait(a)) -> ([]<> crit(a))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial1, ([]<> wait(b)) -> ([]<> crit(b))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(a)) -> ([]<> crit(a))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true

Maude> red modelCheck(initial2, ([]<> wait(b)) -> ([]<> crit(b))) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 4 system states.
result Bool: true
```

Counterexamples

- A counterexample is a pair consisting of two lists of transitions, where the first corresponds to a finite path beginning in the initial state, and the second describes a loop.
- If we check whether, beginning in the state `initial1`, process b will always be waiting, we get a counterexample:

```
Maude> red modelCheck(initial1, [] wait(b)) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 4 system states.

result ModelCheckResult:
  counterexample({$ [a, wait] [b, wait], 'a-enter}
                {[a, critical] [b, wait], 'a-exit}
                {* [a, wait] [b, wait], 'b-enter} ,
                {[a, wait] [b, critical], 'b-exit}
                {$ [a, wait] [b, wait], 'a-enter}
                {[a, critical] [b, wait], 'a-exit}
                {* [a, wait] [b, wait], 'b-enter})
```

Crossing the river

- A shepherd needs to transport to the other side of a river
  - a wolf,
  - a lamb, and
  - a cabbage.
- He has only a boat with room for the shepherd himself and another item.
- The problem is that in the absence of the shepherd
  - the wolf would eat the lamb, and
  - the lamb would eat the cabbage.

Crossing the river

- The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located.
- Constants `left` and `right` represent the two sides of the river.
- Operation `change` is used to modify the corresponding attributes.
- Rules represent the ways of crossing the river that are allowed by the capacity of the boat.
- Properties define the good and bad states:
  - `success` characterizes the state in which the shepherd and his belongings are in the other side,
  - `disaster` characterizes the states in which some eating takes place.

Crossing the river

```
mod RIVER-CROSSING is
  sorts Side Group .

  ops left right : -> Side [ctor] .
  op change : Side -> Side .
  eq change(left) = right .
  eq change(right) = left .

  ops s w l c : Side -> Group [ctor] .
  op __ : Group Group -> Group [ctor assoc comm] .

  var S : Side .

  rl [shepherd] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

Crossing the river

```
mod RIVER-CROSSING-PROP is
  protecting RIVER-CROSSING .
  including MODEL-CHECKER .
  subsort Group < State .

  op initial : -> Group .
  eq initial = s(left) w(left) l(left) c(left) .

  ops disaster success : -> Prop .

  vars S S' S'' : Side .

  ceq (w(S) l(S) s(S') c(S'') |= disaster) = true if S =/= S' .
  ceq (w(S'') l(S) s(S') c(S) |= disaster) = true if S =/= S' .
  eq (s(right) w(right) l(right) c(right) |= success) = true .
endm
```

Crossing the river

- The model checker only returns paths that are counterexamples of properties.
- To find a safe path we need to find a formula that somehow expresses the negation of the property we are interested in: a counterexample will then witness a safe path for the shepherd.
- If no safe path exists, then it is true that whenever `success` is reached a disastrous state has been traversed before:

```
<> success -> (<> disaster /\ ((~ success) U disaster))
```

  Note that this formula is equivalent to the simpler one

```
<> success -> ((~ success) U disaster)
```

- A counterexample to this formula is a safe path, completed so as to have a cycle.

Crossing the river

```
Maude> red modelCheck(initial,
         <> success -> (<> disaster /\ ((~ success) U disaster))) .

result ModelCheckResult: counterexample(
    {s(left) w(left) l(left) c(left),'lamb}
    {s(right) w(left) l(right) c(left),'shepherd}
    {s(left) w(left) l(right) c(left),'wolf}
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'cabbage}
    {s(right) w(right) l(left) c(right),'shepherd}
    {s(left) w(right) l(left) c(right),'lamb}
    {s(right) w(right) l(right) c(right),'lamb}
    {s(left) w(right) l(left) c(right),'shepherd}
    {s(right) w(right) l(left) c(right),'wolf}
    {s(left) w(left) l(left) c(right),'lamb}
    {s(right) w(left) l(right) c(right),'cabbage}
    {s(left) w(left) l(right) c(left),'wolf},
    {s(right) w(right) l(right) c(left),'lamb}
    {s(left) w(right) l(left) c(left),'lamb})
```

Reflection

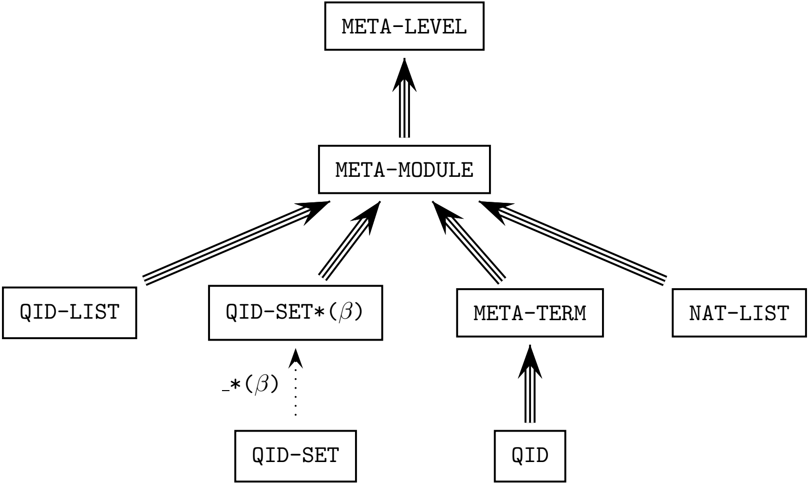
- Rewriting logic is reflective, because there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that:
  - we can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$,
  - then we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

- Since $\mathcal{U}$ is representable in itself, we get a reflective tower

$$\begin{gathered}
\mathcal{R} \vdash t \to t' \\
\Updownarrow \\
\mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\
\Updownarrow \\
\mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\
\vdots
\end{gathered}$$

Maude's metalevel



Maude's metalevel

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` in the module `META-TERM`;
- Maude modules are reified as terms in a data type `Module` in the module `META-MODULE`;
- operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;
- the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
- the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;

Maude's metalevel

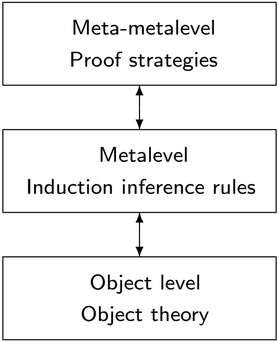
- the process of applying a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
- the process of applying a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
- the process of matching two terms is reified by built-in functions `metaMatch` and `metaXmatch`;
- the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
- parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

Metaprogramming

- Programming at the metalevel: the metalevel equations and rewrite rules operate on representations of lower-level rewrite theories.
- Reflection makes possible many advanced metaprogramming applications, including
  - user-definable strategy languages,
  - language extensions by new module composition operations,
  - development of theorem proving tools, and
  - reifications of other languages and logics within rewriting logic.
- Full Maude extends Maude with special syntax for object-oriented specifications, and with a richer module algebra of parameterized modules and module composition operations
- Theorem provers and other formal tools have underlying inference systems that can be naturally specified and prototyped in rewriting logic. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies.

Developing theorem proving tools

- Theorem-proving tools have a very simple reflective design in Maude.
- The inference system itself may perform theory transformations, so that the theories themselves must be treated as data.
- We need strategies to guide the application of the inference rules.
- Example: Inductive Theorem Prover (ITP).



Full Maude

- The systematic and efficient use of reflection through its predefined `META-LEVEL` module makes Maude remarkably extensible and powerful.
- Full Maude is an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible module algebra of parameterized modules and module composition operations, including parameterized views.
- Full Maude also provides special syntax for object-oriented modules supporting object-oriented concepts such as objects, messages, classes, and multiple class inheritance.

Object-oriented systems

- An object in a given state is represented as a term

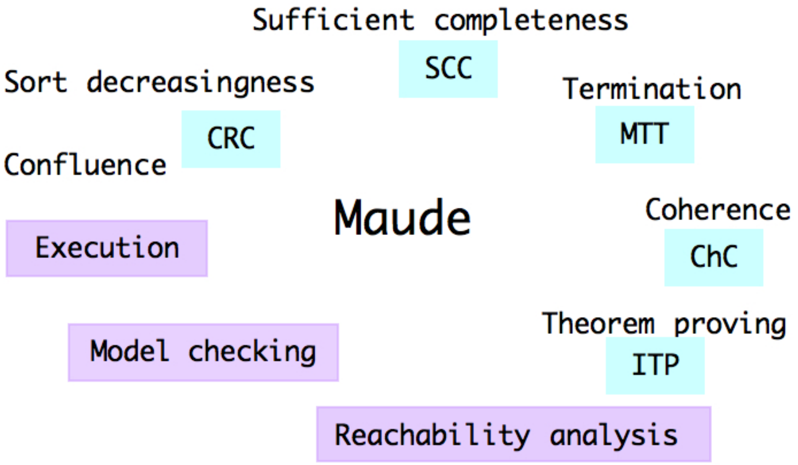
  `< O : C | a1 : v1,..., an : vn >`

  where O is the object's name, belonging to a set `Oid` of object identifiers, C is its class, the `ai`'s are the names of the object's attributes, and the `vi`'s are their corresponding values.
- Messages are defined by the user for each application.
- In a concurrent object-oriented system the concurrent state, which is called a configuration, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms) using rules that describe the effects of communication events between some objects and messages.
- We can regard the special syntax reserved for object-oriented modules as syntactic sugar, because each object-oriented module can be translated into a corresponding system module.

Full Maude

- Full Maude itself can be used as a basis for further extensions, by adding new functionality.
- Full Maude becomes a common infrastructure on top of which one can build other tools:
  - Church-Rosser and coherence checkers for Maude,
  - declarative debuggers for Maude, for wrong and missing answers,
  - Real-Time Maude tool for specifying and analyzing real-time systems,
  - MSOS tool for modular structural operational semantics,
  - Maude-NPA for analyzing cryptographic protocols,
  - strategy language prototype.

Maude's formal environment: tools around Maude



The Maude formal environment

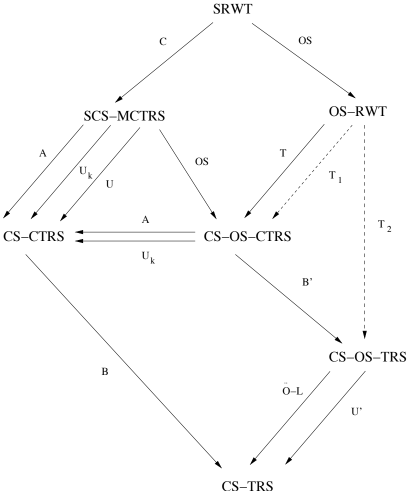
- Besides the built-in support for verifying invariants and LTL formulas, the following tools are also available as part of the Maude formal environment:
  - the Church-Rosser Checker (CRC) can be used to check the Church-Rosser property of functional modules;
  - the Maude Termination Tool (MTT) can be used to prove termination of system modules;
  - the Coherence Checker (ChC) can be used to check the coherence (or ground coherence) of system modules; and
  - the Inductive Theorem Prover (ITP) can be used to verify inductive properties of functional modules;
  - the Sufficient Completeness Checker (SCC) can be used to check that defined functions have been fully defined in terms of constructors.

CRC: a Church-Rosser checker for Maude specifications

- Under the assumption of termination, the Church-Roser Checker tries to check the confluence property.
- The tool
  - may succeed, in which case it responds with a confirmation that the module is confluent, or
  - it may fail to check it, in which case it responds with a set of proof obligations required to ensure that the specification is ground confluent.
- The present CRC tool accepts order-sorted conditional specifications, where each of the operation symbols has either no equational attributes, or any combination of associativity, commutativity, and identity. It is also assumed that such specifications
  - do not contain any built-in function,
  - do not use the otherwise attribute, and
  - that they have already been proved (operationally) terminating.

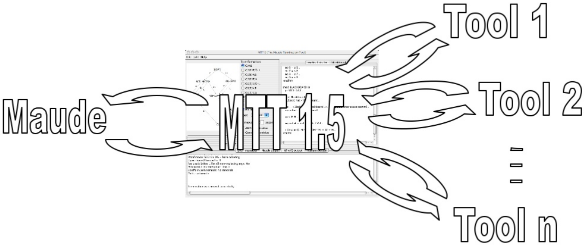
MTT: a Termination Tool for Maude specifications

- A transformational aproach:



MTT: a Termination Tool for Maude specifications

- The MTT implements those transformations between specifications in Maude.
- It also includes a Java GUI to interact with external termination tools, such as AProVE, MU-TERM, etc.
- The termination competition has been useful in two respects: the TPDB syntax, and the interaction with the tools.
- The interaction between the MTT and the termination tools can be either local, or remote via web services.



The ITP: an Inductive Theorem Prover

- The Maude Inductive Theorem Prover tool (ITP) is a theorem-proving assistant.
- It can be used to interactively verify inductive properties of membership equational specifications, or, more precisely, for proving properties of the initial algebra T_Ω of a MEL specification Ω written in Maude.
- It supports proofs by
  - structural induction, and
  - coverset induction.
- It is a program written in Maude such that one can:
  - load in Maude the functional module or modules one wants to reason about,
  - enter named goals to be proved by the ITP, and
  - give commands, corresponding to proof steps.

Real-Time Maude

http://www.ifi.uio.no/RealTimeMaude/

- Real-Time Maude supports the specification of real-time rewrite theories as timed modules and object-oriented timed modules.
- It is written as an extension of Full Maude.
- It trades decidability for expressiveness.
- Formal analyses supported:
  - symbolic simulation,
  - breadth-first search of failures of safety properties, and
  - model-checking of time-bounded temporal properties.
- Complete for discrete time, incomplete in general.
- Succesfully used to analyze large case studies.

Real-time rewrite theories

- A real-time rewrite theory is a rewrite theory with:
  - A specification of a data sort `Time` specifying the time domain.
  - Designated sorts `System` and `GlobalSystem`, and a free constructor

    `{_} : System -> GlobalSystem`

    (for `System` the sort of the state of the system).
  - tick rules, to model time elapse in a system, and
  - "ordinary" rewrite rules, to model instantaneous change.
- `Time` must be specified as a conmutative monoid $(Time, 0, +, <)$.
- Tick rules are rules of the form

$$l : \{t\} \xrightarrow{\tau} \{t'\} \textbf{ if } cond$$

with τ a term of sort `Time`.

Real-time rewrite theories

- Real-time rewrite theories are specified as timed modules or object-oriented timed modules.
- The user has complete freedom to specify the desired data type of time values: discrete or dense, linear or nonlinear, ...
- Tick rules are in general non-deterministic and non-executable.
- Real-Time Maude executes such rules using a time sampling strategy specified by the user.
- The whole time domain is not covered; only some moments are visited.
- Timed object-oriented modules are an extension of both Full Maude's object-oriented modules and timed modules.

Executing and analyzing timed modules

- Three main ways of analyzing timed modules:
  - Rewriting taking time into account.
  - Searching all behaviors relative to a sampling strategy.
  - Temporal logic model checking of properties relative to a sampling strategy.
- Real-Time Maude provides time-bounded model checking and untimed model checking.
- Because of the sampling strategy, the model checker may find that a formula holds in a dense domain even though there exists behaviors that do not satisfy it.
- The underlying logic used to be untimed linear time temporal logic (LTL), but recent extensions have added support for (fragments of) metric LTL and timed CTL.

Strategy language: context

- Maude is a declarative specification and programming language based on rewriting logic.
- A Maude rewrite theory essentially consists of a signature, a set of equations, and a set of rewrite rules.
- Equations are assumed to be confluent and terminating, so that every term has a normal form with respect to equational reduction, but rewrite rules can be nonconfluent or nonterminating.
- Some control when the specifications become executable is required, because the user needs to make sure that the rewriting process does not go in undesired directions.
- The Maude system provides `rewrite` and `frewrite` commands for getting an execution path, and a `search` command for exploring all possible execution paths from a starting term.

Strategy language: main ideas

- But there is a very general additional possibility of being interested in the results of only some execution paths satisfying some constraints. Strategies are needed for this.
- A strategy is described as an operation that, when applied to a given term, produces a set of terms as a result, given that the process is nondeterministic in general.
- A basic strategy consists in applying a rule to a given term, but rules can be conditional with respect to some rewrite conditions which in turn can also be controlled by means of strategies.
- Those basic strategies are combined by means of several operations.
- Furthermore, strategies can also be generic.

Strategy language: semantics and applcations

- This strategy language has been endowed with both a simple set-theoretic semantics and a rewriting semantics, which are sound and complete with respect to the controlled rewrites.
- The first applications of the strategy language included the operational semantics of process algebras such as CCS and the ambient calculus.
- Also the two-level operational semantics of the parallel functional programming language Eden and to modular structural operational semantics.
- It has also been used in the implementation of basic Knuth-Bendix completion algorithms and of congruence closure algorithms.
- Other applications include a sudoku solver, neural networks, membrane systems, hierarchical design graphs, and a representation of BPMN.

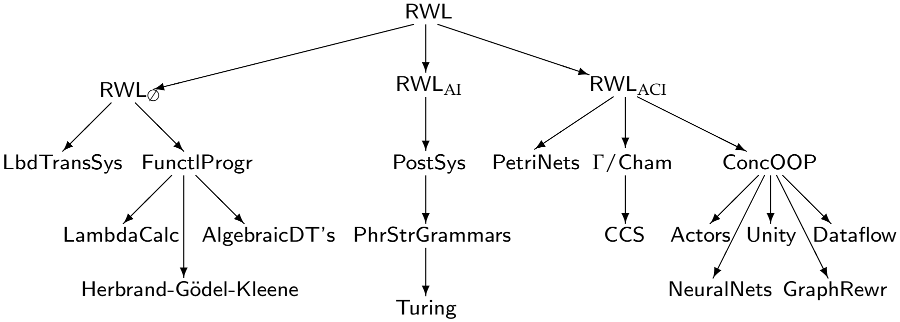
Advertising

Application areas

Application areas

- Models of concurrent computation
  - Equational programming
  - Lambda calculi, including parallel versions
  - Petri nets
  - CCS, LOTOS, and π-calculus
  - Gamma and CHAM (Chemical Abstract Machine)
  - Actors and concurrent objects
  - Neural networks and dataflow languages
- Operational semantics of languages
  - Structural operational semantics (SOS)
  - Agent languages
  - Active networks languages
  - Mobile Maude
  - Hardware description languages
  - Synchronous languages, like PLEXIL

Unification of models of computation



Application areas

- Logical framework and metatool
  - Linear logic
  - Translations between HOL and Nuprl theorem provers
  - Pure type systems
  - Open calculus of constructions
  - Tile logic
- Distributed architectures and components
  - UML diagrams and metamodels
  - Middleware architecture for composable services
  - Reference Model for Open Distributed Processing
  - Validation of OCL properties
  - Model management and model transformations

Application areas

- Specification and analysis of communication protocols
    - Active networks
    - Wireless sensor networks
    - FireWire leader election protocol
    - Design of new protocols, like the L3A accounting protocol
- Modeling and analysis of security properties
    - Cryptographic protocol specification language CAPSL
    - MSR security specification formalism
    - Maude-NPA
    - Network security, like DoS-resilience
    - Browser security
- Real-time, biological, probabilistic systems
    - Real-Time Maude Tool
    - Pathway Logic
    - PMaude

From a satisfied user

In any case, I'd like to say thank you for the great job you have been doing with Full Maude. I find it to be incredibly useful. I've used Full Maude to model a distributed virtual memory system for TCP/IP networks, and there's a pretty good chance that this model will turn into real software that becomes part of the product of my employer. I have known Maude for a while, but that was the first time I actually used it to approach a real world problem. I was surprised how simple and straightforward the process turned out to be. I had a working prototype that exposed all tricky design decisions within less than a week. I've modeled software in Haskell before, and quite liked it, but I have to say that Full Maude is the best system I know so far. My favorite feature are parameterized views. Please know that your efforts are appreciated.

More satisfied users

I'm happy to inform you that with my coworker Marc Nieper-Wisskirchen, we successfully used your Maude program to implement the vertex algebra of operators on the cohomology of Hilbert schemes of points on surfaces. We obtained new results on the characteristic classes of some bundles. Our paper is published in the Journal on Mathematics and Computations (London Math. Soc.) and can be accessed at the following address:
http://www.lms.ac.uk/jcm/10/lms2006-045/
I hope this can be of some interest for you!
Best regards,
Samuel Boissiere
Universite de Nice, France

Structural operational semantics

- In general, an inference rule of the form $\dfrac{S_1 \dots S_n}{S_0}$ can be mapped into a rewrite rule of the form

$$S_1 \dots S_n \longrightarrow S_0 \qquad \text{or} \qquad S_0 \longrightarrow S_1 \dots S_n$$

  that rewrites multisets of judgements S_i.

- In the operational semantics case, it is better to map an inference rule of the form

$$\frac{P_1 \to Q_1 \quad \dots \quad P_n \to Q_n}{P_0 \to Q_0}$$

  to a conditional rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \mathit{if} \quad P_1 \longrightarrow Q_1 \wedge \dots \wedge P_n \longrightarrow Q_n,$$

  where the condition includes rewrites.

Executable semantic framework

- The gap between theory and practice was bridged in Alberto Verdejo's PhD thesis and papers with several case studies:
    - functional language (evaluation and computation semantics, including an abstract machine),
    - imperative language (evaluation and computation semantics),
    - nondeterministic language (computation semantics),
    - Kahn's functional language Mini-ML (evaluation or natural semantics),
    - Milner's CCS (with strong and weak transitions),
    - Full LOTOS (including ACT ONE data type specifications).
- The same techniques were used by other authors for Milner's π-calculus and other languages.

JavaFAN (Java Formal ANalysis)

- Executable rewriting logic semantics of both Java and JVM Bytecode (except for the libraries).
- To keep the framework user-friendly, JavaFAN wraps the Maude specifications and accepts Java or JVM code from the user as input.
- The formal semantic specifications become interpreters to run Java programs on the source code level and/or on the bytecode level.
- Using the underlying features of Maude, JavaFAN can be used to
  - symbolically execute multithreaded programs,
  - detect safety violations searching through an unbounded state space, and
  - verify finite state programs by explicit state model checking.
- JavaFAN's efficiency compares well with other Java analysis tools.
- One of the reasons for efficiency: use of equations instead of rules to express the semantics of deterministic features.

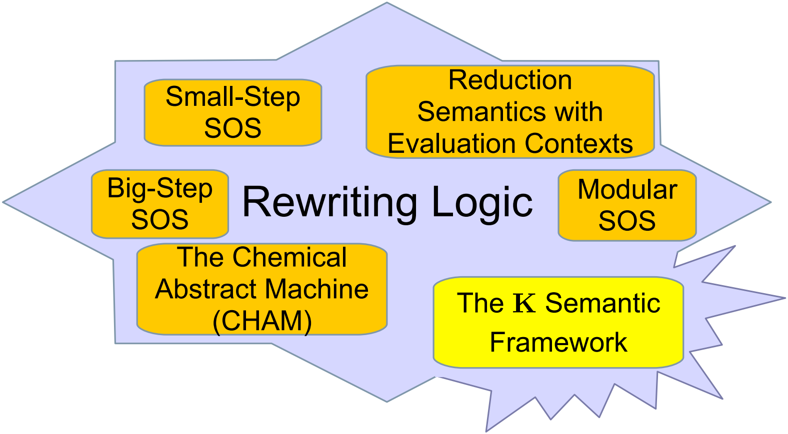
Rewriting logic semantics project

- "The broad goal of the project is to develop a tool-supported computational logic framework for modular programming language design, semantics, formal analysis and implementation, based on rewriting logic."
- Some fundamental references:
  - J. Meseguer and G. Rosu, Rewriting logic semantics: from language specifications to formal analysis tools, *IJCAR 2004*, Springer LNCS 3097.
  - J. Meseguer and G. Rosu, The rewriting logic semantics project, *Theoretical Computer Science*, 2007.
  - T. F. Serbanuta, G. Rosu, and J. Meseguer, A rewriting logic approach to operational semantics, *Information and Computation*, 2009.

“Ecumenical” approach

- Embedding operational semantics styles in rewriting logic
- “Each of these language definitional styles can be faithfully captured as an RLS theory: there is a one-to-one correspondence between computational steps in the original language definition and computational steps in the corresponding RLS theory”
  - Big-step operational semantics (natural semantics)
  - Small-step operational semantics (transition semantics)
  - Modular structural operational semantics (MSOS)
  - Reduction semantics with evaluation contexts
  - Chemical abstract machine
  - First-order continuation-based semantics
- “RLS does not force or pre-impose any given language definitional style, and its exibility and ease of use makes RLS an appealing framework for exploring new definitional styles.”

K framework


Small-Step SOS
Reduction Semantics with Evaluation Contexts
Big-Step SOS
Rewriting Logic
Modular SOS
The Chemical Abstract Machine (CHAM)
The K Semantic Framework

K framework: ambitious features

- Methodology to define languages . . .
- . . . and type checkers, abstract interpreters, domain-specific checkers, etc.
- Arbitrarily complex language features
- Modular (crucial for scalability and reuse)
- Generic (multi-language and multi-paradigm)
- Support for non-determinism and concurrency
- Efficient executability
- State-exploration capabilities (e.g., finite-state model-checking)
- Formal semantics

K basics: computations

- K is based around concepts from Rewriting Logic Semantics, with some intuitions from Chemical Abstract Machines (CHAMs) and Reduction Semantics (RS).
- Abstract computational structures contain context needed to produce a future computation (like continuations).
- Computations take place in the context of a configuration.
- Configurations are hierarchical (like in RLS), made up of K cells.
- Each cell holds specific piece of information: computation, environment, store, etc.
- Two regularly used cells:
  - $\top$ (*top*), representing entire configuration
  - k, representing current computation
- Cells can be repeated (e.g., multiple computations in a concurrent language).

K basics: equations and rules

- Cell k is made up of a list of computational tasks separated by $\curvearrowright$, like $t_1 \curvearrowright t_2 \curvearrowright ... \curvearrowright t_n$.
- Intuition from CHAMs: language constructs can heat (break apart into pieces for evaluation) and cool (form back together).
- Represented using $\rightleftharpoons$, like $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$.
- A heating/cooling pair can be seen as an equation.
- Intuition: $\square$ can be seen as similar to evaluation contexts, marking the location where evaluation can occur.
- Computations are defined used equations and rules.
- Heating/cooling rules (structural equations): manipulate term structure, non-computational, reversible, can think of as just equations.
- Rules: computational, not reversible, may be concurrent.

Example: Cink

- Cink is a kernel of C:
  - functions
  - int expressions
  - simple input/output
  - basic flow control (if, if-else, while, sequential composition)
- Used for teaching purposes.
- Many other examples can be found on Google Code site for K project:
  `http://code.google.com/p/k-framework/`
- There is an online interface of the $\mathbb{K}$ tool
  `https://fmse.info.uaic.ro/tools/K/`

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

- Consider for example the annotated syntax definition for plus:

$$Exp ::= Exp + Exp \ \textcolor{red}{[\text{strict}]}$$

- The strict attribute means the evaluation order is strict and non-deterministic
- This is achieved by two kinds of rules (similar to CHAM):
  - heating rules:

$$E_1 + E_2 \rightarrow E_1 \curvearrowright \square + E_2$$
$$E_1 + E_2 \rightarrow E_2 \curvearrowright E_1 + \square$$

  - cooling rules:

$$I_1 \curvearrowright \square + E_2 \rightarrow I_1 + E_2$$
$$I_2 \curvearrowright E_1 + \square \rightarrow E_1 + I_2$$

$\mathbb{K}$ computations and $\mathbb{K}$ syntax

Computations

- Extend the programming language syntax with a "task sequentialization" operation
    - $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$, where t_i are computational tasks
- Computational tasks: pieces of syntax (with holes), closures, ...

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$$
\begin{array}{llrl}
\mathit{Exp} ::= \mathit{Id} & & & \\
\quad \mid \ \mathit{Exp} + \mathit{Exp} & & [\text{strict}] & \mathit{ERed} + \mathit{ERed} \ \rightarrow \ \mathit{ERed} \curvearrowright \square + \mathit{ERed} \\
\quad \mid \ \mathit{Exp} = \mathit{Exp} & & [\text{strict(2)}] & E = \mathit{ERed} \ \rightarrow \ \mathit{ERed} \curvearrowright E = \square \\
\mathit{Stmt} ::= \mathit{Exp}\ ; & & [\text{strict}] & \mathit{ERed}\ ; \ \rightarrow \ \mathit{ERed} \curvearrowright \square\ ; \\
\quad \mid \ \mathit{Stmt}\ \ \mathit{Stmt} & & [\text{seqstrict}] & \mathit{SRed}\ \ S \ \rightarrow \ \mathit{SRed} \curvearrowright \square\ \ S
\end{array}
$$

Heating syntax through strictness rules

Computation

$\mathsf{y = x{+}2\ ;\ \ x = 7\ ;}$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$		
$\mid\ Exp + Exp$	[strict]	$ERed + ERed\ \rightarrow\ ERed \curvearrowright \square + ERed$
$\mid\ Exp = Exp$	[strict(2)]	$E = ERed\ \rightarrow\ ERed \curvearrowright E = \square$
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ;\ \rightarrow\ ERed \curvearrowright \square\ ;$
$\mid\ Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S\ \rightarrow\ SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

Computation

$\mathsf{y = x{+}2\ ;} \curvearrowright \square\ \mathsf{x = 7\ ;}$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$				
$\mid\ Exp + Exp$	[strict]	$ERed + ERed$	$\rightarrow$	$ERed \curvearrowright \square + ERed$
$\mid\ Exp = Exp$	[strict(2)]	$E = ERed$	$\rightarrow$	$ERed \curvearrowright E = \square$
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ;$	$\rightarrow$	$ERed \curvearrowright \square\ ;$
$\mid\ Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S$	$\rightarrow$	$SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

Computation

$$y = x{+}2 \curvearrowright \square; \curvearrowright \square\ x = 7\ ;$$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$			
$\mid\ Exp + Exp$	[strict]	$ERed + ERed \to ERed \curvearrowright \square + ERed$	
$\mid\ Exp = Exp$	[strict(2)]	$E = ERed \to ERed \curvearrowright E = \square$	
$Stmt ::= Exp\ ;$	[strict]	$ERed\ ; \to ERed \curvearrowright \square\ ;$	
$\mid\ Stmt\ Stmt$	[seqstrict]	$SRed\ S \to SRed \curvearrowright \square\ S$	

Heating syntax through strictness rules

Computation

$$\color{red}{x} + 2 \curvearrowright y = \square \curvearrowright \square; \curvearrowright \square\ x = 7\ ;$$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$Exp ::= Id$			
$\mid$ $Exp + Exp$	[strict]	$ERed + ERed \rightarrow ERed \curvearrowright \square + ERed$
$\mid$ $Exp = Exp$	[strict(2)]	$E = ERed \rightarrow ERed \curvearrowright E = \square$
$Stmt ::= Exp$;	[strict]	$ERed\ ; \rightarrow ERed \curvearrowright \square\ ;$
$\mid$ $Stmt\ \ Stmt$	[seqstrict]	$SRed\ \ S \rightarrow SRed \curvearrowright \square\ \ S$

Heating syntax through strictness rules

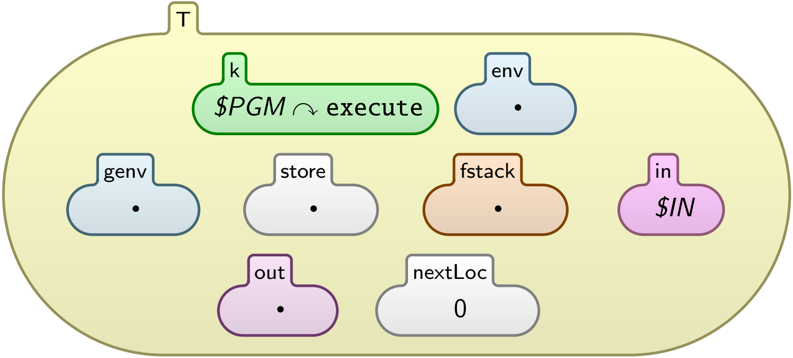
Computation

$x \curvearrowright \square + 2 \curvearrowright y = \square \curvearrowright \square; \curvearrowright \square \; x = 7 \;;$

$\mathbb{K}$ Syntax: BNF syntax annotated with strictness

$\mathit{Exp} ::= \mathit{Id}$		
$\mid \mathit{Exp} + \mathit{Exp}$	[strict]	$\mathit{ERed} + \mathit{ERed} \rightarrow \mathit{ERed} \curvearrowright \square + \mathit{ERed}$
$\mid \mathit{Exp} = \mathit{Exp}$	[strict(2)]	$E = \mathit{ERed} \rightarrow \mathit{ERed} \curvearrowright E = \square$
$\mathit{Stmt} ::= \mathit{Exp}\ ;$	[strict]	$\mathit{ERed}\ ; \rightarrow \mathit{ERed} \curvearrowright \square\ ;$
$\mid \mathit{Stmt}\ \mathit{Stmt}$	[seqstrict]	$\mathit{SRed}\ S \rightarrow \mathit{SRed} \curvearrowright \square\ S$

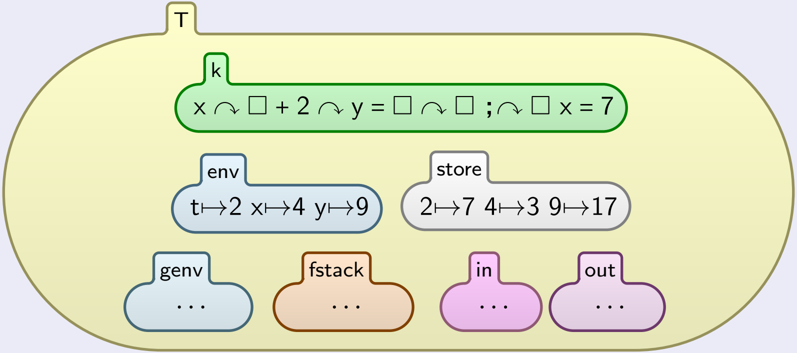
Cink Configuration



𝕂 rules: expressing natural language into rules

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

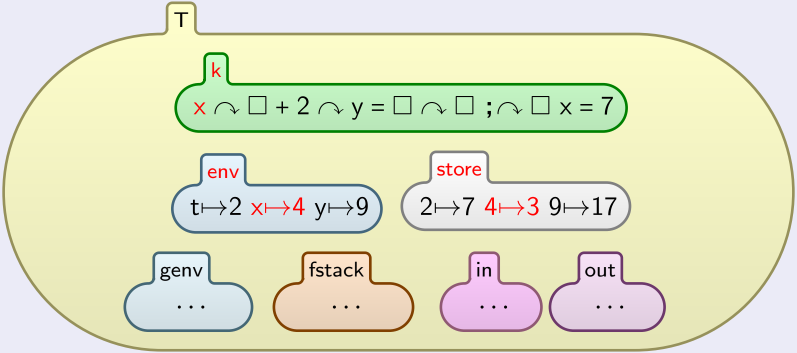


$\mathbb{K}$ rules: expressing natural language into rules

Focusing on the relevant part

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

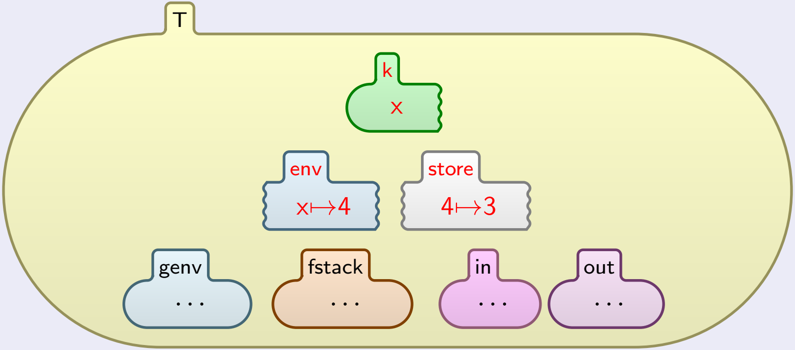


$\mathbb{K}$ rules: expressing natural language into rules

Unnecessary parts of the cells are abstracted away

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

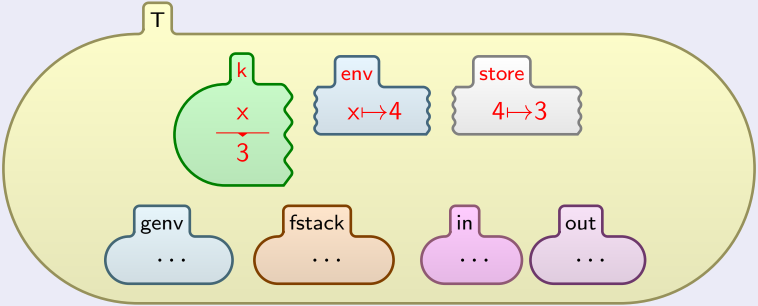


$\mathbb{K}$ rules: expressing natural language into rules

Underlining what to replace, writing the replacement under the line

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

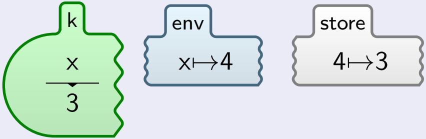


$\mathbb{K}$ rules: expressing natural language into rules

Configuration abstraction: Keep only the relevant cells

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

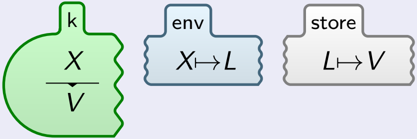


$\mathbb{K}$ rules: expressing natural language into rules

Generalize the concrete instance

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.



This is called context abstraction

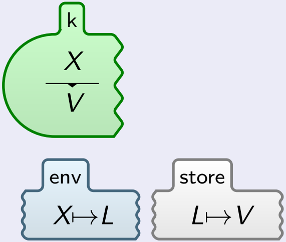
$\mathbb{K}$ rules: expressing natural language into rules

Voilà!

Reading from store via environment

If a variable X is the next thing to be processed and
if X is mapped to a location L in the environment and L to a value V in the store
then process X, replacing it by V.

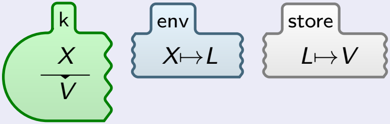
graphic

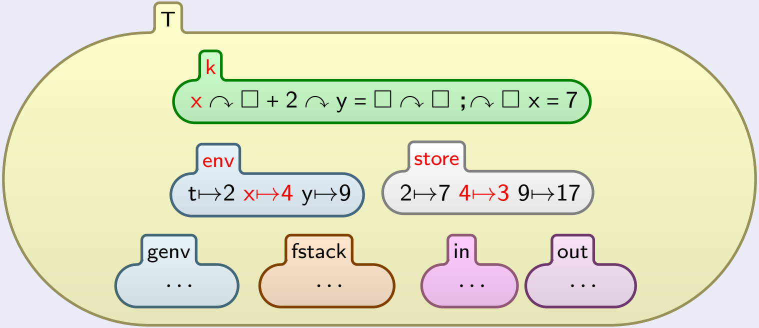


ASCII

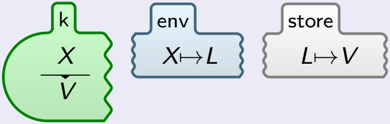
```
rule [lookup]:
     <k> X => V ...</k>
     <env>...  X |-> L ...</env>
     <store>...  L |-> V ...</store>
```

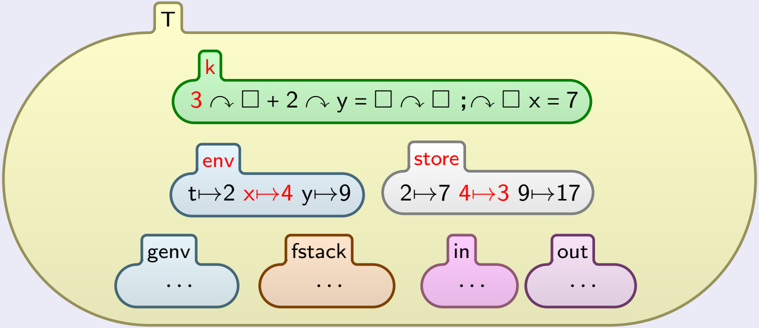
Rules at Work





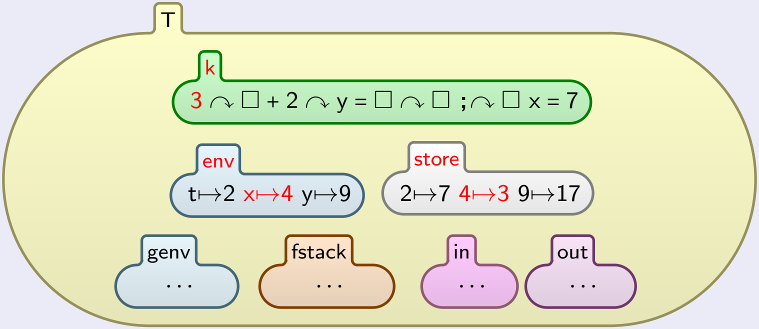
Rules at Work





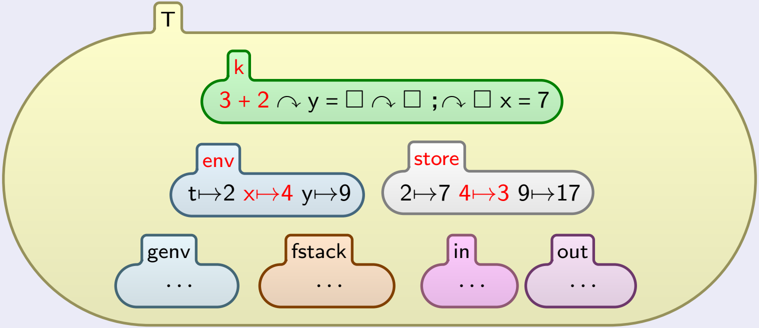
Rules at Work

$ERed \curvearrowright \square + ERed \rightarrow ERed + ERed$



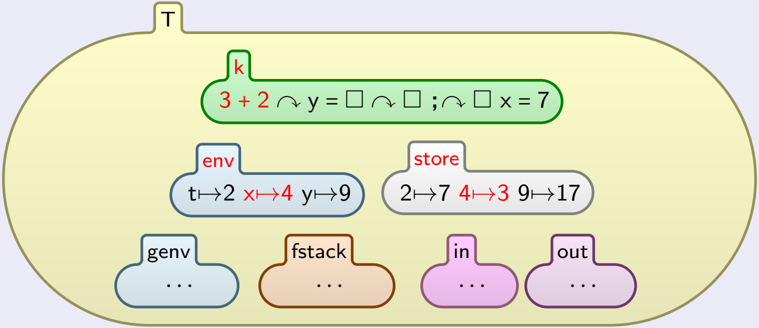
Rules at Work

$$ERed \curvearrowright \square + ERed \rightarrow ERed + ERed$$



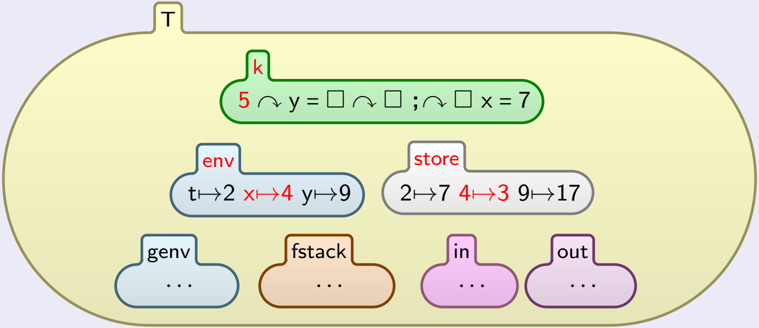
Rules at Work

$I1 + I2 \rightarrow I1 +_{Int} I2$



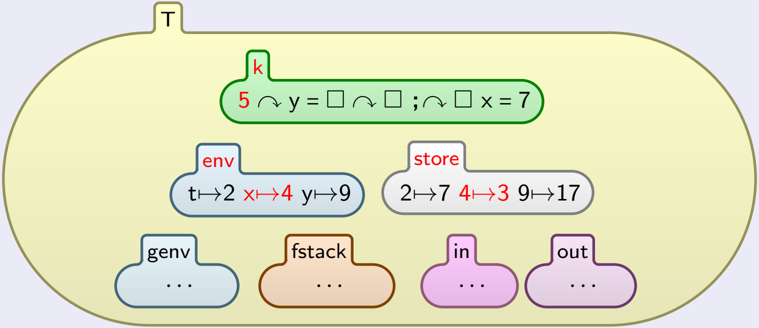
Rules at Work

$I1 + I2 \rightarrow I1 +_{Int} I2$



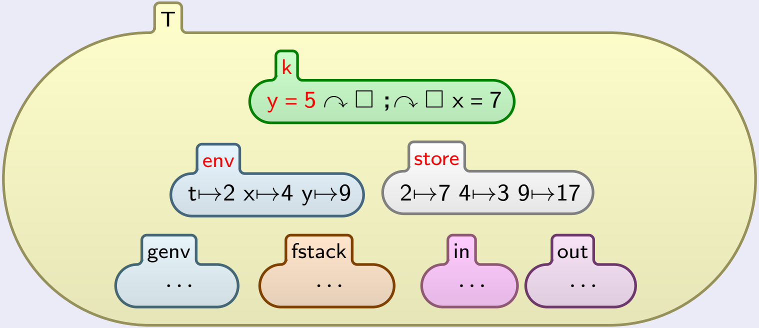
Rules at Work

$$ERed \curvearrowright E = \square \rightarrow E = ERed$$

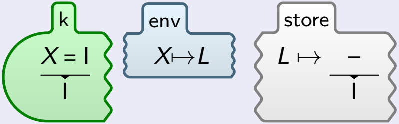


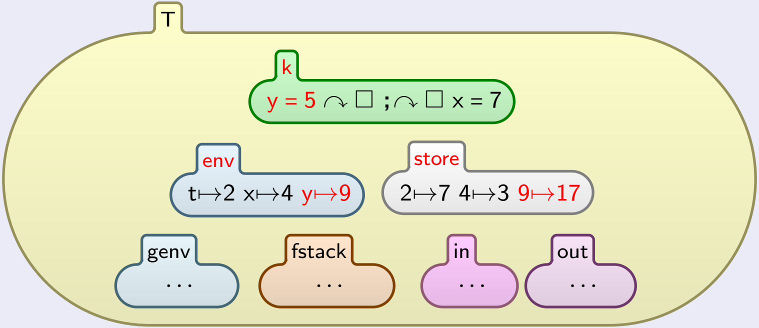
Rules at Work

$ERed \curvearrowright E = \square \rightarrow E = ERed$

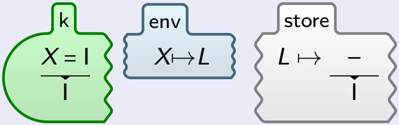


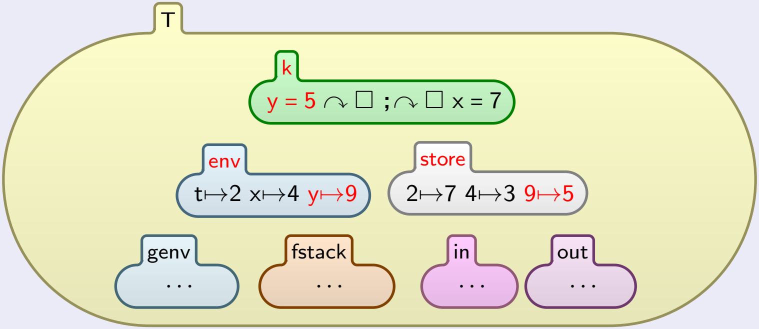
Rules at Work



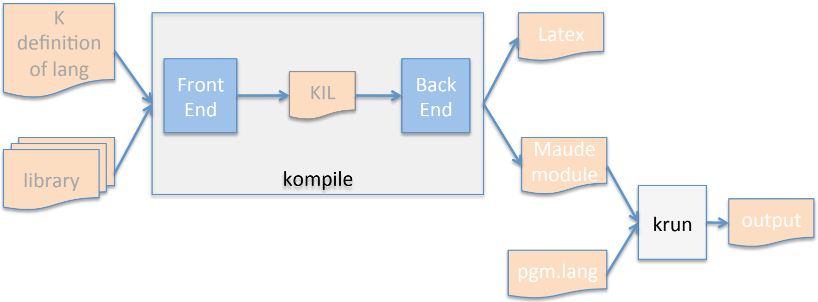


Rules at Work





K tool


K definition of lang
library
Front End
KIL
Back End
kompile
Latex
Maude module
pgm.lang
krun
output

K tool features

- Under continuous development (so still moving target!)
- Compilation of K definitions into Maude and $\LaTeX$
- Literate programming: nicely structured printing of semantics definitions
- Efficient and interactive execution (prototype interpreters)
- Analysis through state-space exploration (from Maude):
  - search
  - model checking
- Further options to control nondeterminism

K framework: some accomplishments

- Embeddings of different operational semantics styles in K.
- Semantics of many languages, toy and real.
- Complete semantics for C (recent paper at POPL 2012).
- Implementation in Maude: K tool and K-Maude.
- Static checking of units of measurement in C.
- Runtime verification of C memory safety.
- Definition of type systems and type inference.
- Compilation of language definitions into competitive interpreters (in OCaml).

Unification

- Given terms t and u, we say that t and u are unifiable if there is a substitution σ such that $\sigma(t) \equiv \sigma(u)$.
- Given an equational theory A and terms t and u, we say that t and u are unifiable modulo A if there is a substitution σ such that $\sigma(t) \equiv_A \sigma(u)$.
- Maude 2.4 supports at the core level and at the metalevel order-sorted equational unification modulo combinations of `comm` and `assoc comm` attributes as well as free function symbols.

Narrowing

- A term t narrows to a term t' using a rule $l \Rightarrow r$ in R and a substitution σ if
  1. there is a subterm $t|_p$ of t at a nonvariable position p of t such that l and $t|_p$ are unifiable via σ, and
  2. $t' = \sigma(t[r]_p)$ is obtained from $\sigma(t)$ by replacing the subterm $\sigma(t|_p) \equiv \sigma(l)$ with the term $\sigma(r)$.
- Narrowing can also be defined modulo an equational theory A.
- Full Maude (2.4 and later) supports a version of narrowing modulo with simplification, where each narrowing step with a rule is followed by simplification to canonical form with the equations.
- There are some restrictions on the allowed rules; for example, they cannot be conditional.

Narrowing reachability analysis

Narrowing can be used as a general deductive procedure for solving reachability problems of the form

$$(\exists \vec{x})\ t_1(\vec{x}) \rightarrow t'_1(\vec{x})\ \wedge \ldots \wedge\ t_n(\vec{x}) \rightarrow t'_n(\vec{x})$$

in a given rewrite theory.

- The terms t_i and t'_i denote sets of states.
- For what subset of states denoted by t_i are the states denoted by t'_i reachable?
- No finiteness assumptions about the state space.
- Sound and complete for topmost rewrite theories.

Maude-NPA

- Maude-NPA (NRL Protocol Analyzer) is a tool to find or prove the absence of attacks using backwards search.
- It analyzes infinite state systems:
  - Active Dolev-Yao intruder,
  - No abstraction or approximation of nonces,
  - Unbounded number of sessions.
- Intruder and honest protocol transitions are represented by a variant of strand space model: strands with a marker denoting the current state.
  - Searches backwards through strands from final state.
  - Set of rewrite rules governs how search is conducted.
  - Sensitive to past and future.

Maude-NPA

- Maude-NPA supports as equations the algebraic properties of the cryptographic functions used in the given protocol:
  - explicit encryption and decryption,
  - exclusive or,
  - modular exponentiation,
  - homomorphism.
- Reasoning modulo such algebraic properties is very important, since it is well-known that some protocols that can be proved secure under the standard Dolev-Yao model, in which the cryptographic functions are treated as a "black box," can actually be broken by an attacker that makes clever use of the algebraic properties of the cryptographic functions of the protocol.

Maude-NPA

- Use rewriting logic as general theoretical framework:
  - protocols and intruder rules are specified as rewrite rules,
  - crypto properties as oriented equational properties and axioms.
- Use narrowing modulo equational theories in two ways:
  - as a symbolic reachability analysis method,
  - as an extensible equational unification method.
- Combine with state reduction techniques of NRL Protocol Analyzer (grammars, optimizations, etc.) by C. Meadows.
- Implement in Maude programming environment:
  - rewriting logic provides theoretical framework and understanding,
  - Maude implementation provides tool support.

Basic structure of Maude-NPA

- Each local execution, or session of a honest principal is represented by a sequence of positive and negative terms called a strand.
  - Negative terms stand for received messages
  - Positive terms stand for sent messages
  - Example: $[\ pke(B, N_A; A)^+,\ pke(A, N_A; N_B)^-,\ pke(B, N_B)^+\ ]$
- Each intruder computation is also represented by a strand
  - Example: $[X^-, pke(A, X)^+]$ and $[X^-, Y^-, (X; Y)^+]$
- Modified strand notation: a marker denotes the current state
  - Example: $[\ pke(B, N_A; A)^+ \mid pke(A, N_A; N_B)^-,\ pke(B, N_B)^+\ ]$
- Sensitive to past and future.
- No data or nonce abstraction.

Basic structure of Maude-NPA

To execute a protocol, associate to it a rewrite theory on sets of strands. $\mathcal{I}$ informally denotes the set of terms known by the intruder, and K the facts known or unknown by the intruder. Then,

- $[\ L \mid M^-, L'\ ]\ \&\ \{M \in \mathcal{I}, K\} \to [\ L, M^- \mid L'\ ]\ \&\ \{M \in \mathcal{I}, K\}$
  Moves input messages into the past.
- $[\ L \mid M^+, L'\ ]\ \&\ \{K\} \to [\ L, M^+ \mid L'\ ]\ \&\ \{K\}$
  Moves output messages that are not read into the past.
- $[\ L \mid M^+, L'\ ]\ \&\ \{M \notin \mathcal{I}, K\} \to [\ L, M^+ \mid L'\ ]\ \&\ \{M \in \mathcal{I}, K\}$
  Joins output message with term in intruder knowledge.

Some work in progress

- Connecting Maude to HETS, heterogeneous verification system developed at Bremen, Germany, which is already connected to theorem provers like Isabelle and Vampire.
- Connecting Maude to SMT solvers.
- Semantics of modeling, real-time, and hardware languages.
- Modeling of cyberphysical systems (avionics, medical systems, ...).
- Security applications.
- More and better equational unification algorithms.
- Conditional narrowing.
- Temporal logic of rewriting.
- Matching logic on top of K framework.
- Multicore reimplementation of Maude.

Many thanks

- Maude team:
  - José Meseguer
  - Francisco Durán
  - Steven Eker
  - Carolyn Talcott
  - Pat Lincoln
  - Manuel Clavel
- Very helpful colleagues:
  - Santiago Escobar and Francisco Durán
  - Grigore Roșu, Dorel Lucanu and Traian Șerbănuță
  - Alberto Verdejo and Miguel Palomino
- ISR 2012 organizers:
  - Salvador Lucas and his team

ISR 2012 sponsors