An introduction to Maude

Narciso Martí-Oliet

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

JAIST-FSSV2010, March 2010

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 $f : s_1 \dots s_n \longrightarrow s$.
- 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

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

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

Maude functional modules

```
fmod BOOLEAN is
  sort Bool .

  op true : -> Bool [ctor] .
  op false : -> Bool [ctor] .

  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  op _or_ : Bool Bool -> Bool .

  var A : Bool .

  eq not true = false .
  eq not false = true .
  eq true and A = A .
  eq false and A = false .
  eq true or A = true .
  eq false or A = A .
endfm
```

Semantics

- A many-sorted (S, Σ)-algebra **A** consists of:
  - a carrier set A_s for each sort $s \in S$, and
  - a function $A_f : A_{s_1} \times \cdots \times A_{s_n} \longrightarrow A_s$ for each operation symbol $f : s_1 \ldots s_n \longrightarrow s$.
- The meaning $[\![t]\!]_{\mathbf{A}}$ of a term t in an algebra **A** is inductively defined.
- An algebra **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 **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 are identified by this congruence if and only if they have the same meaning in all algebras in the loose semantics.

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

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

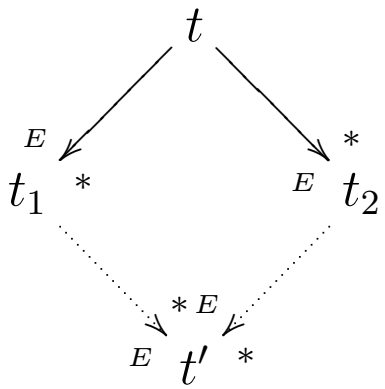
- Given a term $t \in \mathcal{T}_\Sigma(X)$, the pattern, and a subject ground term $u \in \mathcal{T}_\Sigma$, we say that t matches u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.

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 $t \rightarrow_E t'$.
- It can be proved that if $t \rightarrow_E t'$ then $[\![t]\!]_{\mathbf{A}} = [\![t']\!]_{\mathbf{A}}$ for any algebra $\mathbf{A}$ satisfying E.
- We write $t \rightarrow_E^* t'$ to mean either $t = t'$ (0 steps) or $t \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \cdots \rightarrow_E t_n \rightarrow_E t'$ 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 unified by further rewriting: if $t \to_E^* t_1$ and $t \to_E^* t_2$, then there exists a term t' such that $t_1 \to_E^* t'$ and $t_2 \to_E^* t'$.



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

Confluence and termination

- 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.
- 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.

```
Maude> reduce in BOOLEAN :
         (false and true) or (true and (false or true)) .
result Bool: true

Maude> reduce in BOOLEAN :   not (not false or not true) .
result Bool: false
```

Natural numbers

```
fmod UNARY-NAT is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

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

- Can we add the equation

  ```
  eq M + N = N + M .
  ```

  expressing commutativity of addition?

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.

```
fmod NAT3 is
  including UNARY-NAT .
  var N : Nat .
  eq s(s(s(N))) = N .
endfm
```

Operations on natural numbers

```
fmod NAT+OPS is
  protecting BOOLEAN .
  protecting UNARY-NAT .

  ops _*_ _-_ : Nat Nat -> Nat .
  ops _<=_ _>_ : Nat Nat -> Bool .

  vars N M : Nat .
  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 M > N = not (M <= N) .
endfm
```

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.

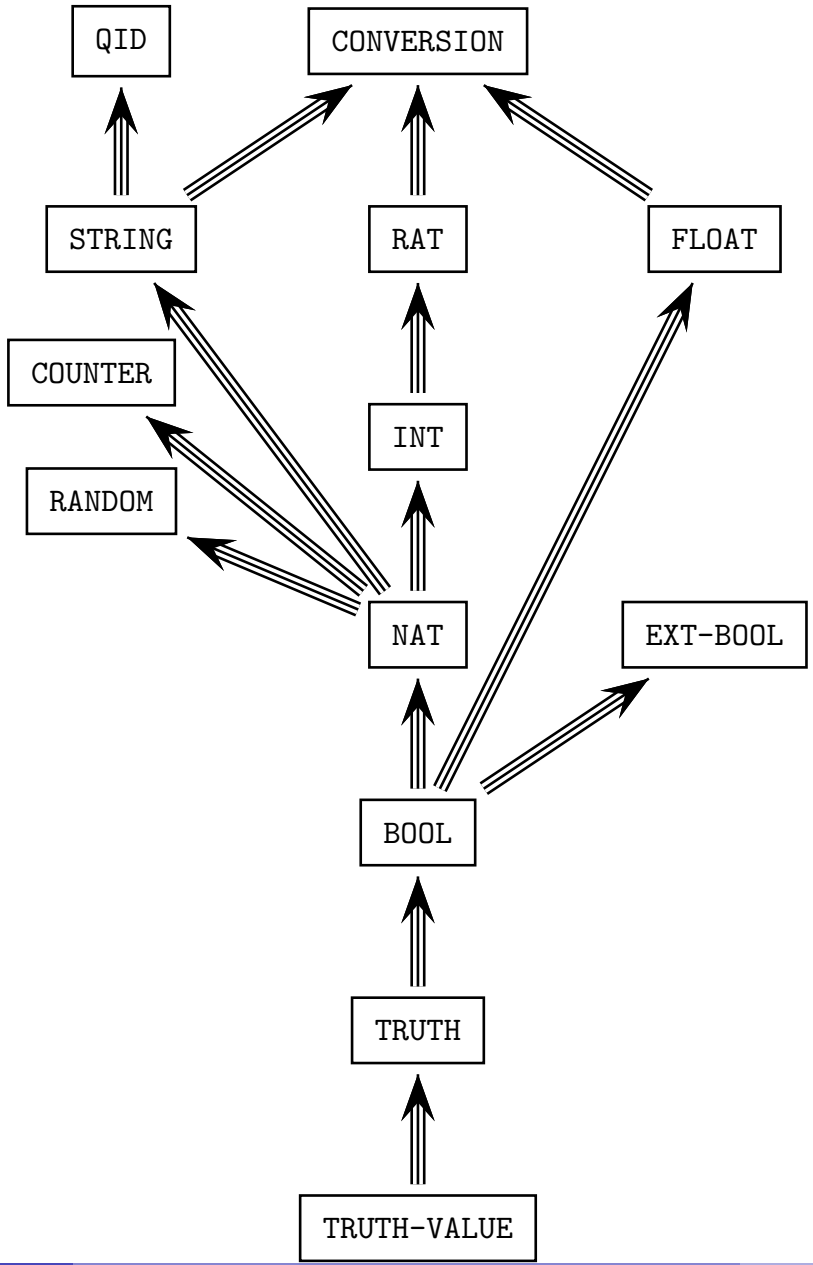
Natural numbers division

```
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



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

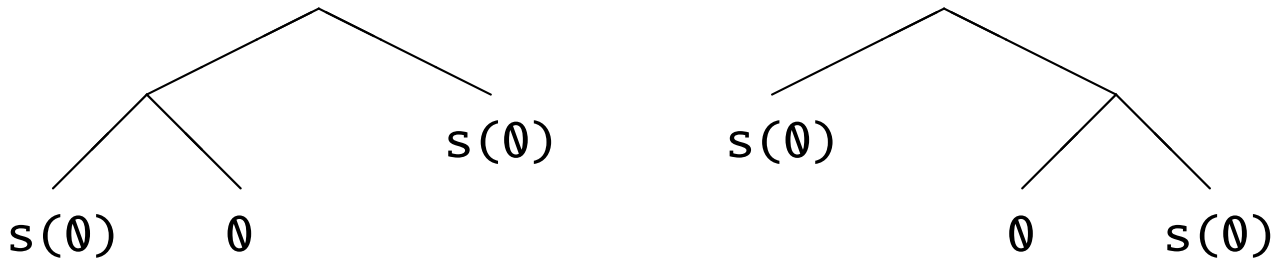
```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .

  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:

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

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. In the example of natural numbers division, the term

  ```
  s(s(s(0))) div (s(s(0)) - s(0))
  ```

  is not even well formed, because the subterm `s(s(0)) - s(0)` has least sort `Nat`, while the `div` operation expects 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.

$$(\forall X) \;\; t = t' \;\; \text{if} \;\; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

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

Membership equational logic 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

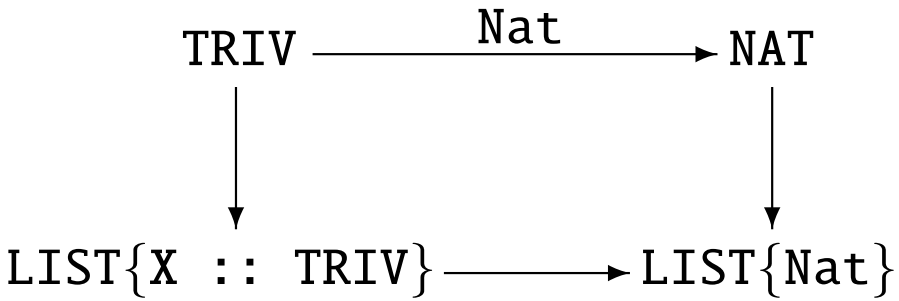
  ```
  fmod LIST{X :: TRIV} is  ...  endfm
  ```

  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`.



- 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 $t \longrightarrow t'$ 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 $r : [t]_E \longrightarrow [t']_E$ 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:

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

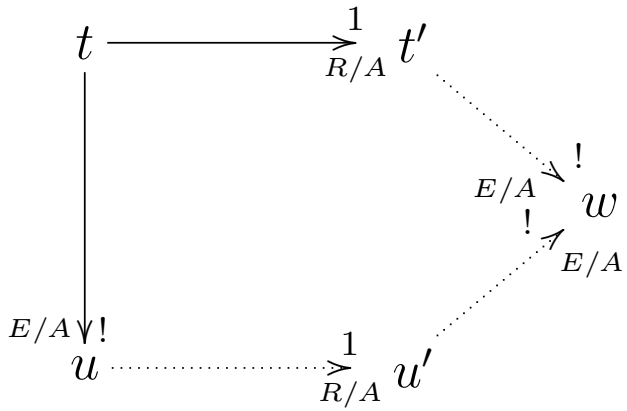
System modules

- 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

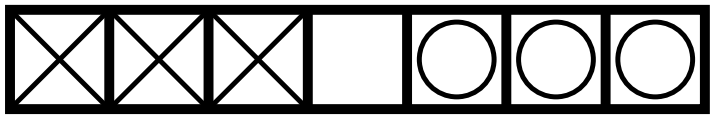
- The rules R in the module must be coherent with the equations E modulo A, allowing us to intermix rewriting with rules and rewriting with equations without losing rewrite computations by failing to perform a rewrite that would have been possible before an equational deduction step was taken.



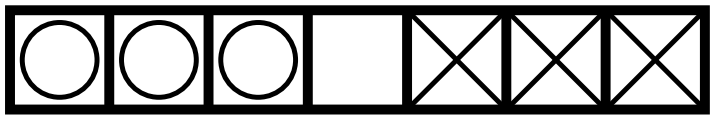
- A simple strategy available in these circumstances is to always reduce to canonical form using E before applying any rule in R.
- In this way, we get the effect of rewriting modulo $E \cup A$ with just a matching algorithm for A.

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
```

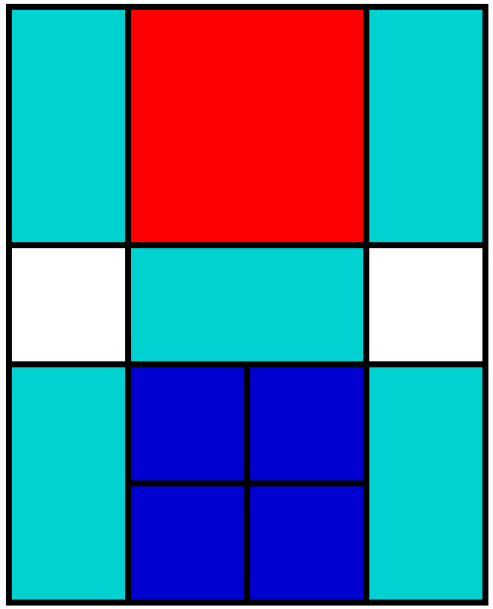
Crossing the bridge

- 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.

The Khun Phan puzzle



- Can we move the big square to where the small ones are?
- Can we reach a completely symmetric configuration?

The Khun Phan puzzle

```
mod KHUN-PHAN is
  protecting NAT .
  sorts Piece Board .
  subsort Piece < Board .

  *** each piece carries the coordinates of its upper left corner
  ops empty bigsq smallsq hrect vrect : Nat Nat -> Piece .

  *** board is nonempty multiset of pieces
  op __ : Board Board -> Board [assoc comm] .

  op initial : -> Board .

  eq initial
    = vrect(1, 1)        bigsq(2, 1)          vrect(4, 1)
      empty(1, 3)        hrect(2, 3)          empty(4, 3)
      vrect(1, 4) smallsq(2, 4) smallsq(3, 4) vrect(4, 4)
                  smallsq(2, 5) smallsq(3, 5) .
```

The Khun Phan puzzle

```
vars X Y : Nat .

rl [sqr] : smallsq(X, Y) empty(s(X), Y)
  => empty(X, Y) smallsq(s(X), Y) .
rl [sql] : smallsq(s(X), Y) empty(X, Y)
  => empty(s(X), Y) smallsq(X, Y) .
rl [squ] : smallsq(X, s(Y)) empty(X, Y)
  => empty(X, s(Y)) smallsq(X, Y) .
rl [sqd] : smallsq(X, Y) empty(X, s(Y))
  => empty(X, Y) smallsq(X, s(Y)) .

rl [Sqr] : bigsq(X, Y) empty(s(s(X)), Y) empty(s(s(X)), s(Y))
  => empty(X, Y) empty(X, s(Y)) bigsq(s(X), Y) .
rl [Sql] : bigsq(s(X), Y) empty(X, Y) empty(X, s(Y))
  => empty(s(s(X)), Y) empty(s(s(X)), s(Y)) bigsq(X, Y) .
rl [Squ] : bigsq(X, s(Y)) empty(X, Y) empty(s(X), Y)
  => empty(X, s(s(Y))) empty(s(X), s(s(Y))) bigsq(X, Y) .
rl [Sqd] : bigsq(X, Y) empty(X, s(s(Y))) empty(s(X), s(s(Y)))
  => empty(X, Y) empty(s(X), Y) bigsq(X, s(Y)) .
```

The Khun Phan puzzle

```
  rl [hrectr] : hrect(X, Y) empty(s(s(X)), Y)
    => empty(X, Y) hrect(s(X), Y) .
  rl [hrectl] : hrect(s(X), Y) empty(X, Y)
    => empty(s(s(X)), Y) hrect(X, Y) .
  rl [hrectu] : hrect(X, s(Y)) empty(X, Y) empty(s(X), Y)
    => empty(X, s(Y)) empty(s(X), s(Y)) hrect(X, Y) .
  rl [hrectd] : hrect(X, Y) empty(X, s(Y)) empty(s(X), s(Y))
    => empty(X, Y) empty(s(X), Y) hrect(X, s(Y)) .

  rl [vrectr] : vrect(X, Y) empty(s(X), Y) empty(s(X), s(Y))
    => empty(X, Y) empty(X, s(Y)) vrect(s(X), Y) .
  rl [vrectl] : vrect(s(X), Y) empty(X, Y) empty(X, s(Y))
    => empty(s(X), Y) empty(s(X), s(Y)) vrect(X, Y) .
  rl [vrectu] : vrect(X, s(Y)) empty(X, Y)
    => empty(X, s(s(Y))) vrect(X, Y) .
  rl [vrectd] : vrect(X, Y) empty(X, s(s(Y)))
    => empty(X, Y) vrect(X, s(Y)) .
endm
```

The Khun Phan puzzle

- With the following command we get all possible 964 final configurations to the game:

```
Maude> search initial =>* B:Board bigsq(2, 4) .
```

- The final state used, `B:Board bigsq(2,4)`, represents any final situation such that the upper left corner of the big square is at coordinates $(2,4)$.
- The `search` command does not enumerate the different ways of reaching the same configuration.
- The shortest path leading to the final configuration, due to the breadth-first search, reveals that it consists of 112 moves:

```
Maude> show path labels 23721 .
```

The Khun Phan puzzle

- The following command shows that it is not possible to reach a position symmetric to the initial one.

```
Maude> search initial
         =>* vrect(1, 1) smallsq(2, 1) smallsq(3, 1) vrect(4, 1)
                         smallsq(2, 2) smallsq(3, 2)
             empty(1, 3)         hrect(2, 3)         empty(4, 3)
             vrect(1, 4)         bigsq(2, 4)         vrect(4, 4) .

No solution.
```

Reflection

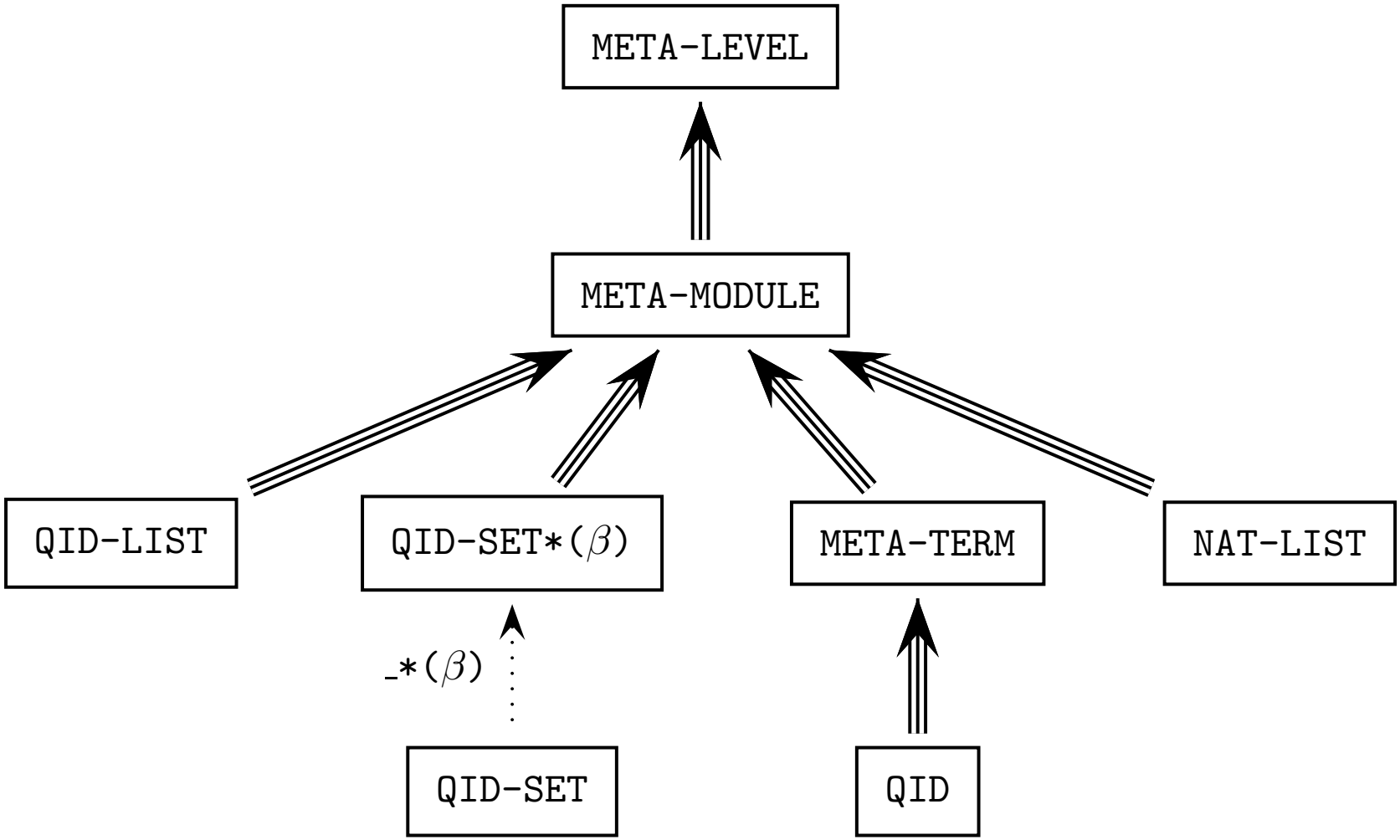
- Rewriting logic is reflective, because there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that:
  - we can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$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

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

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

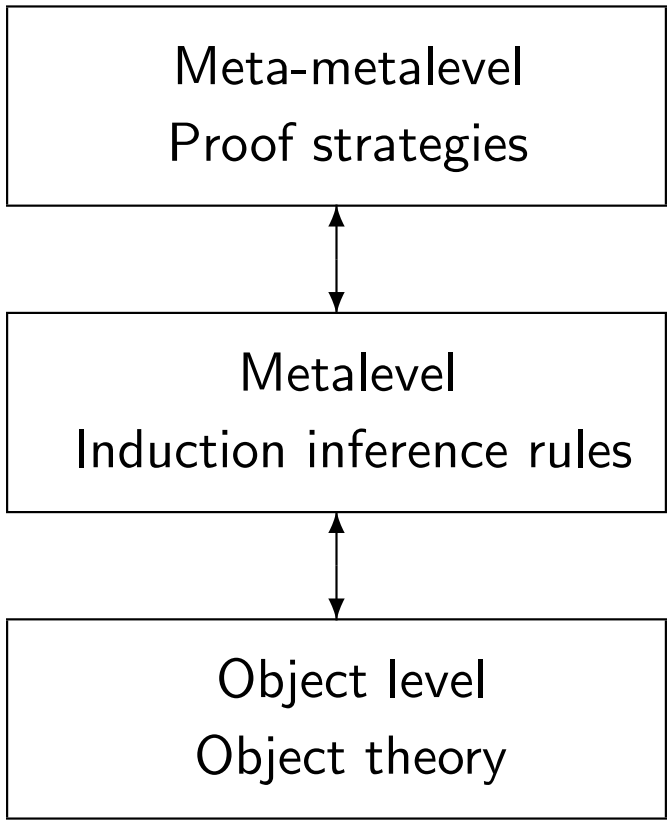
- the process of applying a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
- the process of applying a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
- the process of matching two terms is 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.

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 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

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

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.

Application areas

- Models of concurrent computation
  - Equational programming
  - Lambda calculi
  - Petri nets
  - CCS and π-calculus
  - Actors
- Operational semantics of languages
  - Structural operational semantics (SOS)
  - Agent languages
  - Active networks languages
  - Mobile Maude
  - Hardware description languages

Application areas

- Logical framework and metatool
  - Linear logic
  - Translations between HOL and NuprI 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
- Modeling and analysis of security protocols
  - Cryptographic protocol specification language CAPSL
  - MSR security specification formalism
  - Maude-NPA
- 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

Some work in progress

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

The Book


All About Maude – A High-Performance Logical Framework
This monograph gives a comprehensive account of Maude, a language and system based on rewriting logic. Maude and its formal tool environment can be used in three mutually reinforcing ways: as a declarative programming language, as an executable formal specification language, and as a formal verification system. Maude is used in many institutions around the world for teaching, research, and formal modeling and analysis of concurrent and distributed systems.
Many examples are used throughout the book to illustrate the main ideas, features, and uses of Maude. The book comes with a CD-ROM containing the complete Maude 2.3 software distribution (including source code), a pdf version of this monograph, and the executable Maude code for all the examples in the book.
Clavel et al.
Tutorial
LNCS 4350
Manuel Clavel Francisco Durán
Steven Eker Patrick Lincoln
Narciso Martí-Oliet José Meseguer
Carolyn Talcott
All About Maude – A High-Performance Logical Framework
How to Specify, Program and Verify Systems in Rewriting Logic
MAUDE CT
W MAUDE AVE 900
Springer
with CD-ROM
LNCS 4350
All About Maude – A High-Performance Logical Framework
with CD-ROM
In parallel to the printed book, each new volume is published electronically in LNCS Online.
Detailed information on LNCS can be found at
www.springer.com/lncs
Proposals for publication should be sent to
LNCS Editorial, Tiergartenstr. 17, 69121 Heidelberg, Germany
E-mail: lncs@springer.com
ISSN 0302-9743
ISBN 978-3-540-71940-3
9 783540 719403
Lecture Notes in Computer Science
LNCS LNAI LNBI
springer.com

Many thanks

- JAIST-FSSV2010 organizers:
  - Kokichi Futatsugi
- Maude team:
  - José Meseguer
  - Francisco Durán
  - Steven Eker
  - Manuel Clavel
  - Carolyn Talcott
  - Pat Lincoln