

Specifying and proving in Maude

Narciso Martí-Oliet

Manuel Clavel

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
narciso|clavel@sip.ucm.es

Málaga, ESLLI 2006

Summary

Maude is a high-level language and high-performance system supporting both equational and rewriting computation for a wide range of applications. In the first part of this course, we provide an introduction to **equational specification** and **rule-based programming** in Maude 2.0, showing the difference between equations and rules. We use typical data structures (stacks, queues, lists, binary trees) and well-known mathematical games and puzzles to illustrate the features and expressive power of Maude, emphasizing the full generality with which **membership equational logic** and **rewriting logic** are supported. Indeed, the expressive version of equational logic in which Maude is based, namely membership equational logic, 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.

Summary

Moreover, exploiting the fact that rewriting logic is reflective, a key distinguishing feature of Maude is its systematic and efficient use of **reflection**, a feature that makes Maude remarkably extensible and powerful, and that allows many advanced **metaprogramming and metalanguage applications**. In the second part of the course we describe the use of an **inductive theorem prover**, ITP (which itself is developed and integrated in Maude by means of the powerful metalevel and metalanguage features offered by the latter, as mentioned above) to prove properties of the data types specified in Maude and to verify algorithms that are based on them.

Introduction

<http://maude.cs.uiuc.edu>

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- **Membership equational logic** improves order-sorted algebra.
- **Rewriting logic** is a logic of concurrent change.

Introduction

<http://maude.cs.uiuc.edu>

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- **Membership equational logic** improves order-sorted algebra.
- **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.

Introduction

<http://maude.cs.uiuc.edu>

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- **Membership equational logic** improves order-sorted algebra.
- **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.

Contents

- ① Introduction
- ② Functional modules
 - Many-sorted equational specifications
 - Order-sorted equational specifications
 - Equational attributes
 - Membership equational logic specifications
- ③ Parameterization
 - Theories and views
 - Data structures
- ④ System modules
 - Rewriting logic
 - Playing with Maude
 - Lambda calculus

Contents

- ⑤ Full Maude
 - Object-oriented systems
- ⑥ Model checking
 - LTL and Kripke structures
 - Mutual exclusion
 - Crossing the river
- ⑦ Reflection
 - Maude's metalevel
 - Some descent functions
 - Internal strategies
 - Metaprogramming

Many-sorted equational specifications

- Algebraic specifications are used to declare different kinds of data together with the operations that act upon them.
- The behavior of operations is described by means of **(conditional) equations**.
- It is useful to distinguish two kinds of operations: **constructors**, that are used to construct or generate the data, and the remaining operations (that can also be classified as **modifiers** or as **observers**).
- 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 an $S^* \times S$ -sorted family $\Sigma = \{\Sigma_{\bar{s}, s} \mid \bar{s} \in S^*, s \in S\}$ of sets of operation symbols.
- When $f \in \Sigma_{\bar{s}, s}$, we say that f has **rank** $\langle \bar{s}, s \rangle$, **arity** (or argument sorts) \bar{s} , and **coarity** (or value sort, or range sort) s .

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 an $S^* \times S$ -sorted family $\Sigma = \{\Sigma_{\bar{s}, s} \mid \bar{s} \in S^*, s \in S\}$ of sets of operation symbols.
- When $f \in \Sigma_{\bar{s}, s}$, we say that f has **rank** $\langle \bar{s}, s \rangle$, **arity** (or argument sorts) \bar{s} , and **coarity** (or value sort, or range sort) s .
- With the declared operations we can construct **terms** to denote the data being specified.
- Terms are typed, and can have **variables**.

Terms and equations

- 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** $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is inductively defined by the following conditions:
 - $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
 - $\Sigma_{\varepsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
 - If $f \in \Sigma_{\bar{s},s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ ($i = 1, \dots, n$), where $\bar{s} = s_1 \dots s_n \neq \varepsilon$, then $f(t_1, \dots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$.

Terms and equations

- 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** $\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$ is inductively defined by the following conditions:
 - $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
 - $\Sigma_{\varepsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
 - If $f \in \Sigma_{\bar{s},s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ ($i = 1, \dots, n$), where $\bar{s} = s_1 \dots s_n \neq \varepsilon$, then $f(t_1, \dots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$.
- A **Σ -equation** is an expression $(\bar{x} : \bar{s}) l = r$, where $\bar{x} : \bar{s}$ is a (finite) set of variables, and l and r are terms in $\mathcal{T}_{\Sigma,s}(\bar{x} : \bar{s})$ for some sort s .
- A **conditional Σ -equation** is an expression

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

where $(\bar{x} : \bar{s}) l = r$ and $(\bar{x} : \bar{s}) u_i = v_i$ are Σ -equations.

- A **many-sorted specification** consists of a signature (S, Σ) and a set E of (conditional) Σ -equations.

Semantics

- A many-sorted (S, Σ) -algebra \mathbf{A} consists of a carrier set A_s for each $s \in S$ and a function $A_f^{\bar{s}, s} : A_{\bar{s}} \rightarrow A_s$ for each operation symbol $f \in \Sigma_{\bar{s}, s}$.
- It is possible to define inductively the meaning of a term in an algebra and then satisfaction of a (conditional) equation by an algebra.
- The **loose semantics** of a many-sorted specification (S, Σ, E) is the set of (S, Σ) -algebras that satisfy all the (conditional) equations in E .

Semantics

- A many-sorted (S, Σ) -algebra \mathbf{A} consists of a carrier set A_s for each $s \in S$ and a function $A_f^{\bar{s}, s} : A_{\bar{s}} \rightarrow A_s$ for each operation symbol $f \in \Sigma_{\bar{s}, s}$.
- It is possible to define inductively the meaning of a term in an algebra and then satisfaction of a (conditional) equation by an algebra.
- The **loose semantics** of a many-sorted specification (S, Σ, E) is the set of (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}_{Σ} 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.

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 .
```


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
```

Matching, rewriting, and equational simplification

- Given an S -sorted family of variables X for a signature (S, Σ) , a (ground) **substitution** is a sort-preserving map $\sigma : X \rightarrow \mathcal{T}_\Sigma$; such a map extends uniquely to terms $\mathcal{T}_\Sigma(X) \rightarrow \mathcal{T}_\Sigma$.
- Given a term $t \in \mathcal{T}_\Sigma(X)$, 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.

Matching, rewriting, and equational simplification

- Given an S -sorted family of variables X for a signature (S, Σ) , a (ground) **substitution** is a sort-preserving map $\sigma : X \rightarrow \mathcal{T}_\Sigma$; such a map extends uniquely to terms $\mathcal{T}_\Sigma(X) \rightarrow \mathcal{T}_\Sigma$.
- Given a term $t \in \mathcal{T}_\Sigma(X)$, 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.
- In a Σ -equation $(\bar{x} : \bar{s}) l = r$ 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 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 t' 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 $t \rightarrow_E t'$.

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 $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.
- A set of equations E is **terminating** when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \dots$.
- 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.

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 $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.
- A set of equations E is **terminating** when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \dots$.
- 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: $t \downarrow_E \equiv t' \downarrow_E$.
- 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.

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 an equation 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.

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

Conditional equations

- **Equational conditions** in conditional equations are made up of individual equations $t = t'$.
- A condition can be either a single equation or a conjunction of equations using the binary conjunction connective \wedge which is assumed associative.
- Furthermore, the concrete syntax of equations in conditions has three variants:
 - ordinary equations $t = t'$,
 - **matching equations** $t := t'$, and
 - **abbreviated Boolean equations** of the form t , with t a term of sort `Bool`, abbreviating the equation $t = \text{true}$.
- The Boolean terms appearing most often in abbreviated Boolean equations are terms using the built-in equality `_==_` and inequality `_=/=_` predicates (from **predefined module `BOOL`**).

Integers

```
fmod INTEGERS is
  sort Int .

  op 0 : -> Int [ctor] .
  op s : Int -> Int [ctor] .
  op p : Int -> Int [ctor] .

  op _+_ : Int Int -> Int .
  op _-_ : Int Int -> Int .
  op *_ : Int Int -> Int .
  op -_ : Int -> Int .
  op _<=_ : Int Int -> Bool .

  vars N M : Int .

  eq s(p(N)) = N .
  eq p(s(N)) = N .
```

Integers

$$\text{eq } 0 + N = N .$$

$$\text{eq } s(M) + N = s(M + N) .$$

$$\text{eq } p(M) + N = p(M + N) .$$

$$\text{eq } N - 0 = N .$$

$$\text{eq } M - s(N) = p(M - N) .$$

$$\text{eq } M - p(N) = s(M - N) .$$

$$\text{eq } - N = 0 - N .$$

$$\text{eq } 0 * N = 0 .$$

$$\text{eq } s(M) * N = (M * N) + N .$$

$$\text{eq } p(M) * N = (M * N) - N .$$

$$\text{eq } s(M) \leq N = M \leq p(N) .$$

$$\text{eq } p(M) \leq N = M \leq s(N) .$$

$$\text{eq } 0 \leq 0 = \text{true} .$$

$$\text{eq } 0 \leq p(0) = \text{false} .$$

$$\text{ceq } 0 \leq s(M) = \text{true if } 0 \leq M .$$

$$\text{ceq } 0 \leq p(M) = \text{false if not}(0 \leq M) .$$

endfm

Cardinality of finite sets

```
fmod NAT< is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _<_ : Nat Nat -> Bool .

  vars N N' : Nat .

  eq N < 0 = false .
  eq 0 < s N = true .
  eq s N < s N' = N < N' .
endfm
```

Cardinality of finite sets

```
fmod NAT< is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _<_ : Nat Nat -> Bool .
```

```
vars N N' : Nat .
```

```
eq N < 0 = false .
eq 0 < s N = true .
eq s N < s N' = N < N' .
```

```
endfm
```

```
fmod CARDINAL is
  protecting NAT< .
  sort Set .
  op emptySet : -> Set [ctor] .
  op _.._ : Nat Set -> Set [ctor] .   *** add element
  op # : Set -> Nat .                 *** count elements
```

Cardinality of finite sets

```

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

eq N . N . S = N . S .                *** idempotency
ceq N . N' . S = N' . N . S if N' < N . *** commutativity

eq #(emptySet) = 0 .
eq #(N . emptySet) = s 0 .
eq #(0 . s N . emptySet) = s s 0 .
eq #(s N . s N' . emptySet) = #(N . N' . emptySet) .
ceq #(N . N' . S) = s s #(S)
  if #(N . N' . emptySet) = s s 0
  /\ #(N . S) = s #(S)
  /\ #(N' . S) = s #(S) .
endfm

```

Order-sorted equational specifications

- There are operations that are **not defined** for some values, like division on natural numbers.
- We can often avoid the possibility of considering partial functions by extending the many-sorted equational logic to **order-sorted** equational logic, which allows us to define **subsorts** corresponding to the domain of definition of a function, whenever such subsorts can be defined by means of constructors.

Order-sorted equational specifications

- There are operations that are **not defined** for some values, like division on natural numbers.
- We can often avoid the possibility of considering partial functions by extending the many-sorted equational logic to **order-sorted** equational logic, which allows us to define **subsorts** corresponding to the domain of definition of a function, whenever such subsorts can be defined by means of constructors.
- An order-sorted signature adds a partial order relation to the set of sorts S , such that $s \leq s'$ is interpreted semantically by the subset inclusion $A_s \subseteq A_{s'}$ between the corresponding carrier sets in the algebras.
- Moreover, operations can be overloaded:
 - **subsort overloading**: addition both on natural numbers and on integers,
 - **ad-hoc overloading**: the same symbol can be used in unrelated sorts.

Preregularity and sort-decreasingness

- 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.

Preregularity and sort-decreasingness

- 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.
- Another important property is **sort-decreasingness**.
- Assuming E is confluent and terminating, the canonical form $t \downarrow_E$ of a term t by the equations E should have the **least sort possible** among the sorts of all the terms equivalent to it by the equations E ; and it should be possible to compute this least sort from the canonical form itself, using only the operator declarations.
- By a **Church-Rosser and terminating** theory (Σ, E) we precisely mean one that is confluent, terminating, and 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 $0 \leq N = \text{true}$.

eq $s(M) \leq 0 = \text{false}$.

eq $s(M) \leq s(N) = M \leq N$.

eq $N > M = \text{not } (N \leq M)$.

ceq $N \text{ div } P = 0 \text{ if } P > N$.

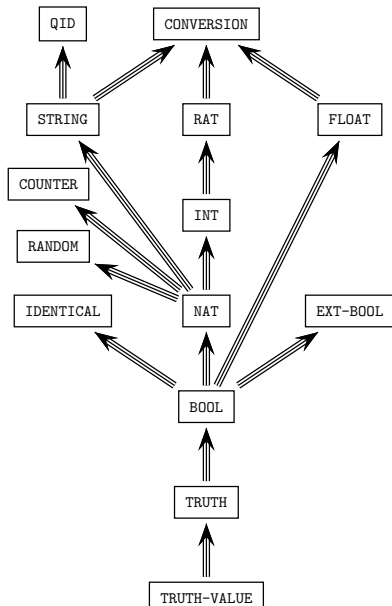
ceq $N \text{ div } P = s((N - P) \text{ div } P) \text{ if } P \leq N$.

ceq $N \text{ mod } P = N \text{ if } P > N$.

ceq $N \text{ mod } P = (N - P) \text{ mod } P \text{ if } P \leq N$.

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 equational axioms in a way that allows Maude to use these equations efficiently in a built-in way.
- Currently Maude supports the following equational attributes:
 - `assoc` (**associativity**),
 - `comm` (**commutativity**),
 - `idem` (**idempotency**),
 - `id`: $\langle Term \rangle$ (**identity**, with the corresponding term for the identity element),
 - `left id`: $\langle Term \rangle$ (**left identity**, with the corresponding term for the left identity element), and
 - `right id`: $\langle Term \rangle$ (**right identity**, with the corresponding term for the right identity element).
- These attributes are only allowed for **binary** operators satisfying some appropriate requirements that depend 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 .

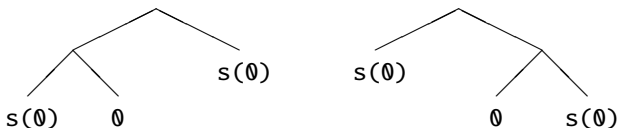
  sorts Tree .
  subsort Nat < Tree .
  op __ : Tree Tree -> Tree [ctor] .
  op depth : Tree -> Nat .
  op width : Tree -> Nat .

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

$$\begin{aligned}
 \text{length}(L \text{ nil}) &= \text{length}(L) + \text{length}(\text{nil}) \\
 &= \text{length}(L \text{ nil}) + \text{length}(\text{nil}) \\
 &= (\text{length}(L) + \text{length}(\text{nil})) + \text{length}(\text{nil}) \\
 &= \dots
 \end{aligned}$$

because of the identification $L = L \text{ nil}$.

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

```

Multisets with otherwise

```

fmod NAT-MSETS-OWISE 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 N in N S = true .
  eq N in S = false [otherwise] .
  eq mult(N, N S) = s(0) + mult(N, S) .
  eq mult(N, S) = 0 [otherwise] .
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:

$$\text{empty-set} = \text{empty-set empty-set} \rightarrow \text{empty-set} \dots$$

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, respectively, because we cannot rely on the order in which equations are applied.
- The operations `_in_` and `delete` can also be defined with `owise`.

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 ordered 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 $s(s(s(0))) \text{ div } (s(s(0)) - s(0))$ is not even well formed.
- The subterm $s(s(0)) - s(0)$ has least sort Nat , while the div operation expects its second argument to be of sort $\text{NzNat} < \text{Nat}$.
- 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.

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

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

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.

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.
- An **operator declaration** like

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

can be understood as a declaration at the kind level together with the conditional membership axiom

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

- A **subsort declaration** $\text{NzNat} < \text{Nat}$ can be understood as the conditional membership axiom

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

Multiples of 3

```
fmod 3*NAT is
  sorts Zero Nat .
  subsort Zero < Nat .

  op 0 : -> Zero [ctor] .
  op s_ : Nat -> Nat [ctor] .

  sort 3*Nat .
  subsorts Zero < 3*Nat < Nat .

  var M3 : 3*Nat .

  mb (s s s M3) : 3*Nat .
endfm
```

Palindromes

```
fmod PALINDROME is
  protecting QID .
  sorts Word Pal .
  subsort Qid < Pal < Word .

  op nil : -> Pal [ctor] .
  op _ : Word Word -> Word [ctor assoc id: nil] .

  var I : Qid .
  var P : Pal .

  mb I P I : Pal .
endfm
```

Ordered lists

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

  sorts OrdList NeOrdList .
  subsort NeOrdList < OrdList NeList < List .

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

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

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

  vars N M : Nat .
  vars L L' : List .
  vars OL OL' : OrdList .
  var NEOL : NeOrdList .
```

Ordered lists

```
mb [] : OrdList .
```

```
mb N : [] : NeOrdList .
```

```
cmb N : NEOL : NeOrdList 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) .
```

Ordered 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
```

Partial constructors: powerlists

- A **powerlist** must be of length 2^n for some $n \in \mathbb{N}$.

```
fmod POWERLIST is
  protecting NAT .

  sort Pow .

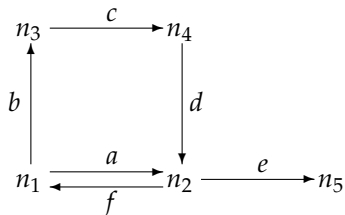
  op [_] : Nat -> Pow [ctor] .
  op |_| : [Pow] [Pow] -> [Pow] [assoc] .
  op len : Pow -> Nat .

  var I : Nat .
  vars P Q : Pow .

  cmb P | Q : Pow if len(P) = len(Q) .

  eq len([I]) = 1 .
  eq len(P | Q) = len(P) + len(Q) .
endfm
```

Partial constructors: paths



fmod A-GRAPH is

sorts Edge Node .

ops n1 n2 n3 n4 n5 : -> Node [ctor] .

ops a b c d e f : -> Edge [ctor] .

ops source target : Edge -> Node .

eq source(a) = n1 . eq target(a) = n2 .

eq source(b) = n1 . eq target(b) = n3 .

eq source(c) = n3 . eq target(c) = n4 .

eq source(d) = n4 . eq target(d) = n2 .

eq source(e) = n2 . eq target(e) = n5 .

eq source(f) = n2 . eq target(f) = n1 .

endfm

Paths

```
fmod PATH is
  protecting NAT .
  protecting A-GRAPH .
  sorts Path .
  subsorts Edge < Path .
  op _;_ : [Path] [Path] -> [Path] [assoc] .
  ops source target : Path -> Node .
  op length : Path -> Nat .

  var E : Edge .
  var P : Path .
  cmb (E ; P) : Path if target(E) == source(P) .

  eq source(E ; P) = source(E) .
  eq target(P ; E) = target(E) .
  eq length(E) = 1 .
  eq length(E ; P) = 1 + length(P) .
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 simplification 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 TOSET< is
  protecting BOOL .
  sort Elt .
  ops _<_ _>_ : Elt Elt -> Bool .
  vars E1 E2 E3 : Elt .
  eq E1 < E1 = false .
  ceq E1 < E3 = true if E1 < E2 and E2 < E3 [nonexec] .
  ceq E1 < E2 or E2 < E1 = true if E1 /= E2 [nonexec] .
  eq E1 > E2 = E2 < E1 .
endfth
```

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

```
fth TOSET<= is
  including TOSET< .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= Y = X < Y or X == Y .
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.
- In many simple cases it is completely obvious:

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

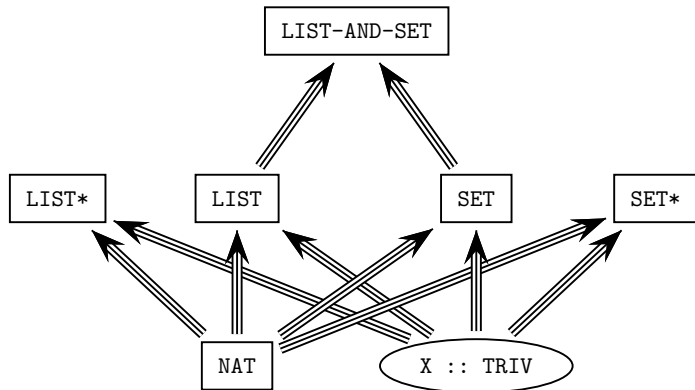
Parameterization: instantiation

- 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.

Predefined parameterized modules



Stacks

```
fmod STACK{X :: TRIV} is
  sorts NeStack{X} Stack{X} .
  subsort NeStack{X} < Stack{X} .
  op empty : -> Stack{X} [ctor] .
  op push : X$Elt Stack{X} -> NeStack{X} [ctor] .
  op pop : NeStack{X} -> Stack{X} .
  op top : NeStack{X} -> X$Elt .
  op isEmpty : Stack{X} -> Bool .

  var S : Stack{X} .
  var E : X$Elt .

  eq pop(push(E,S)) = S .
  eq top(push(E,S)) = E .
  eq isEmpty(empty) = true .
  eq isEmpty(push(E,S)) = false .
endfm
```

Stacks

```
view Int from TRIV to INT is
  sort Elt to Int .
endv
```

```
fmod STACK-TEST is
  protecting STACK{Int} .
endfm
```

```
Maude> red top(push(4,push(5,empty))) .
result NzNat : 4
```


Queues

```

fmod QUEUE{X :: TRIV} is
  sort NeQueue{X} Queue{X} .
  subsort NeQueue{X} < Queue{X} .
  op empty : -> Queue{X} [ctor] .
  op enqueue : Queue{X} X$Elt -> NeQueue{X} [ctor] .
  op dequeue : NeQueue{X} -> Queue{X} .
  op first : NeQueue{X} -> X$Elt .
  op isEmpty : Queue{X} -> Bool .

var Q : Queue{X} .
var E : X$Elt .

eq dequeue(enqueue(empty,E)) = empty .
ceq dequeue(enqueue(Q,E)) = enqueue(dequeue(Q),E) if Q /= empty .
eq first(enqueue(empty,E)) = E .
ceq first(enqueue(Q,E)) = first(Q) if Q /= empty .
eq isEmpty(empty) = true .
eq isEmpty(enqueue(Q,E)) = false .
endfm

```

Priority queues

```
fmod PRIORITY-QUEUE{X :: TOSET<=} is
  sort NePQueue{X} PQueue{X} .
  subsort NePQueue{X} < PQueue{X} .
  op empty : -> PQueue{X} .
  op insert : PQueue{X} X$Elt -> NePQueue{X} .
  op deleteMin : NePQueue{X} -> PQueue{X} .
  op findMin : NePQueue{X} -> X$Elt .
  op isEmpty : PQueue{X} -> Bool .

var PQ : PQueue{X} .
vars E F : X$Elt .
```

Priority queues

```
eq insert(insert(PQ,E),F) = insert(insert(PQ,F),E) [nonexec] .
eq deleteMin(insert(empty,E)) = empty .
ceq deleteMin(insert(insert(PQ,E),F)) =
    insert(deleteMin(insert(PQ,E)),F)
    if findMin(insert(PQ,E)) <= F .
ceq deleteMin(insert(insert(PQ,E),F)) = insert(PQ,E)
    if findMin(insert(PQ,E)) > F .
eq findMin(insert(empty,E)) = E .
ceq findMin(insert(insert(PQ,E),F)) = findMin(insert(PQ,E))
    if findMin(insert(PQ,E)) <= F .
ceq findMin(insert(insert(PQ,E),F)) = F
    if findMin(insert(PQ,E)) > F .
eq isEmpty(empty) = true .
eq isEmpty(insert(PQ,E)) = false .
endfm
```

Priority queues

```
view IntAsToset<= from TOSET<= to INT is
  sort Elt to Int .
endv
```

```
fmod PRIORITY-QUEUE-TEST is
  protecting PRIORITY-QUEUE{IntAsToset<=} .
endfm
```

```
Maude> red findMin(insert(insert(empty,4),5)) .
result NzNat: 4
```

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 ordered lists

```
view Tiset from TRIV to TOSET<= is
  sort Elt to Elt .
endv
```

```
fmod ORD-LIST{X} :: TOSET<=} is
  protecting LIST-CONS{Tiset}{X} .
```

```
sorts OrdList{X} NeOrdList{X} .
subsorts NeOrdList{X} < OrdList{X} < List{Tiset}{X} .
subsort NeOrdList{X} < NeList{Tiset}{X} .
```

```
vars N M : X$Elt .
vars L L' : List{Tiset}{X} .
vars OL OL' : OrdList{X} .
var NEOL : NeOrdList{X} .
```

Parameterized ordered lists

```

mb [] : OrdList{X} .
mb (N : []) : NeOrdList{X} .
cmb (N : NEOL) : NeOrdList{X} if N <= head(NEOL) .

op insertion-sort : List{ToSet}{X} -> OrdList{X} .
op insert-list : OrdList{X} X$Elt -> OrdList{X} .

op mergesort : List{ToSet}{X} -> OrdList{X} .
op merge : OrdList{X} OrdList{X} -> OrdList{X} .

op quicksort : List{ToSet}{X} -> OrdList{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 ordered lists

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

```
fmod ORD-LIST-TEST is
  protecting ORD-LIST{NatAsToset<=} .
endfm
```

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

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

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

Parameterized multisets

```

fmod MULTISSET{X :: TRIV} is
  protecting NAT .
  sort Mset{X} .
  subsort X$Elt < Mset{X} .
  op empty : -> Mset{X} [ctor] .
  op __ : Mset{X} Mset{X} -> Mset{X} [ctor assoc comm id: empty] .

  vars E E' : X$Elt .
  vars S S' : Mset{X} .

  op isEmpty : Mset{X} -> Bool .
  eq isEmpty(empty) = true .
  eq isEmpty(E S) = false .

  op size : Mset{X} -> Nat .
  eq size(empty) = 0 .
  eq size(E S) = 1 + size(S) .

  op mult : X$Elt Mset{X} -> Nat .
  eq mult(E, E S) = 1 + mult(E, S) .
  eq mult(E, S) = 0 [owise] .

```

Parameterized multisets

```
op isIn : X$Elt Mset{X} -> Bool .  
eq isIn(E, E S) = true .  
eq isIn(E, S) = false [owise] .
```

```
op delete : X$Elt Mset{X} -> Mset{X} .  
eq delete(E, E S) = delete(E, S) .  
eq delete(E, S) = S [owise] .
```

```
op delete1 : X$Elt Mset{X} -> Mset{X} .  
eq delete1(E, E S) = S .  
eq delete1(E, S) = S [owise] .
```

```
op intersection : Mset{X} Mset{X} -> Mset{X} .  
eq intersection(E S, E S') = E intersection(S, S') .  
eq intersection(S, S') = empty [owise] .
```

```
op difference : Mset{X} Mset{X} -> Mset{X} .  
eq difference(E S, E S') = difference(S, S') .  
eq difference(S, S') = S [owise] .
```

```
endfm
```

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} .
```

```
  eq left(L [E] R) = L .  
  eq right(L [E] R) = R .  
  eq root(L [E] R) = E .
```

Binary trees

```

op depth : BinTree{X} -> Nat .
ops leaves preorder inorder postorder : BinTree{X} -> List{X} .

eq depth(empty) = 0 .
eq depth(L [E] R) = 1 + max(depth(L), depth(R)) .

eq leaves(empty) = [] .
eq leaves(empty [E] empty) = E : [] .
eq leaves(NEL [E] R) = leaves(NEL) ++ leaves(R) .
eq leaves(L [E] NER) = leaves(L) ++ leaves(NER) .

eq preorder(empty) = [] .
eq preorder(L [E] R) = E : (preorder(L) ++ preorder(R)) .
eq inorder(empty) = [] .
eq inorder(L [E] R) = inorder(L) ++ (E : inorder(R)) .
eq postorder(empty) = [] .
eq postorder(L [E] R) = postorder(L) ++ (postorder(R) ++ (E : []))
endfm

```

Binary search trees

- Search trees, like **dictionaries**, contain in the nodes pairs formed by a key and its associated contents.
- The search tree structure is with respect to a **strict total order on keys**, but contents can be over an arbitrary sort.
- When we insert a pair $\langle K, C \rangle$ and the key K already appears in the tree in a pair $\langle K, C' \rangle$, insertion takes place by **combining** the contents C' and C .
- As part of the requirement parameter theory for the contents we will have an associative binary operator **combine** on the sort **Contents**.

```
fth CONTENTS is
  sort Contents .
  op combine : Contents Contents -> Contents [assoc] .
endfth
```

Binary search trees

- The module for search trees will be imported in the specifications of AVL and red-black trees.
- It is important to be able to **add** new information in the nodes: for AVL trees, the **depth** of the tree hanging in each node, and for red-black trees the appropriate node **color**.

Binary search trees

- The module for search trees will be imported in the specifications of AVL and red-black trees.
- It is important to be able to **add** new information in the nodes: for AVL trees, the **depth** of the tree hanging in each node, and for red-black trees the appropriate node **color**.
- A **record** is defined as a collection (with an associative and commutative union operator denoted by $_ , _$) of pairs consisting of a field name and an associated value.

```
fmod RECORD is
  sorts Record .
  op null : -> Record [ctor] .
  op _ , _ : Record Record -> Record [ctor assoc comm id: null] .
endfm
```

```
view Record from TRIV to RECORD is
  sort Elt to Record .
endv
```


Binary search trees

```
fmod SEARCH-TREE{X :: TOSET<, Y :: CONTENTS} is
  including BIN-TREE{Record} .
  sorts SearchTree{X, Y} NeSearchTree{X, Y} .
  subsorts NeSearchTree{X, Y} < SearchTree{X, Y} < BinTree{Record} .
  subsort NeSearchTree{X, Y} < NeBinTree{Record} .
```

```
--- Construction of the record used as node
op key:_ : X$Elt -> Record [ctor] .
op contents:_ : Y$Contents -> Record [ctor] .
op not-found : -> [Y$Contents] .
op key : Record -> X$Elt .
op contents : Record -> Y$Contents .
```

```
var Rec : Record .
var K : X$Elt .
var C : Y$Contents .
```

```
eq key(Rec, key: K) = K .
eq contents(Rec, contents: C) = C .
```

Binary search trees

--- Definition of binary search trees.

```
ops min max : NeSearchTree{X, Y} -> Record .
```

```
var Rec' : Record .
```

```
vars L R : SearchTree{X, Y} .
```

```
vars L' R' : NeSearchTree{X, Y} .
```

```
var C' : Y$Contents .
```

```
eq min(empty [Rec] R) = Rec .
```

```
eq min(L' [Rec] R) = min(L') .
```

```
eq max(L [Rec] empty) = Rec .
```

```
eq max(L [Rec] R') = max(R') .
```

```
mb empty : SearchTree{X, Y} .
```

```
mb empty [Rec] empty : NeSearchTree{X, Y} .
```

```
cmb L' [Rec] empty : NeSearchTree{X, Y}
```

```
  if key(max(L')) < key(Rec) .
```

```
cmb empty [Rec] R' : NeSearchTree{X, Y}
```

```
  if key(Rec) < key(min(R')) .
```

```
cmb L' [Rec] R' : NeSearchTree{X, Y}
```

```
  if key(max(L')) < key(Rec) and key(Rec) < key(min(R')) .
```

Binary search trees

--- Operations for binary search trees.

op insert : SearchTree{X, Y} X\$Elt Y\$Contents
 -> SearchTree{X, Y} .

op lookup : SearchTree{X, Y} X\$Elt -> [Y\$Contents] .

op delete : SearchTree{X, Y} X\$Elt -> SearchTree{X, Y} .

op find : SearchTree{X, Y} X\$Elt -> Bool .

eq insert(empty, K, C) = empty [key: K,contents: C] empty .

eq insert(L [Rec,key: K,contents: C] R, K, C')
 = L [Rec,key: K, contents: combine(C, C')] R .

ceq insert(L [Rec] R, K, C) = insert(L, K, C) [Rec] R
 if K < key(Rec) .

ceq insert(L [Rec] R, K, C) = L [Rec] insert(R, K, C)
 if key(Rec) < K .

eq lookup(empty, K) = not-found .

eq lookup(L [Rec, key: K,contents: C] R, K) = C .

ceq lookup(L [Rec] R, K) = lookup(L, K) if K < key(Rec) .

ceq lookup(L [Rec] R, K) = lookup(R, K) if key(Rec) < K .

Binary search trees

```
eq delete(empty, K) = empty .
ceq delete(L [Rec] R, K) = delete(L, K) [Rec] R if K < key(Rec) .
ceq delete(L [Rec] R, K) = L [Rec] delete(R, K) if key(Rec) < K .
eq delete(empty [Rec, key: K, contents: C] R, K) = R .
eq delete(L [Rec, key: K, contents: C] empty, K) = L .
eq delete(L' [Rec, key: K, contents: C] R', K)
  = L' [min(R')] delete(R', key(min(R'))) .

eq find(empty, K) = false .
eq find(L [Rec, key: K, contents: C] R, K) = true .
ceq find(L [Rec] R, K) = find(L, K) if K < key(Rec) .
ceq find(L [Rec] R, K) = find(R, K) if key(Rec) < K .
endfm
```

Binary search trees

```
view StringAsContents from CONTENTS to STRING is
  sort Contents to String .
  op combine to _+_ .
endv
```

```
view IntAsToset< from TOSET< to INT is
  sort Elt to Int .
endv
```

Binary search trees

```
view StringAsContents from CONTENTS to STRING is
  sort Contents to String .
  op combine to _+_ .
endv
```

```
view IntAsToset< from TOSET< to INT is
  sort Elt to Int .
endv
```

```
fmod SEARCH-TREE-TEST is
  protecting SEARCH-TREE{IntAsToset<, StringAsContents} .
endfm
```

```
Maude> red insert(insert(empty, 1, "a"), 2, "b") .
result NeSearchTree{IntAsToset<,StringAsContents}:
  empty[key: 1,contents: "a"](empty[key: 2,contents:"b"]empty)
```

```
Maude> red lookup(insert(insert(insert(empty, 1, "a"),
                                2, "b"), 1, "c"), 1) .
result String: "ac"
```

AVL trees

- AVL trees are binary search trees satisfying the additional constraint in each node that the **difference between the depth of both children is at most one**.
- Then the **depth** of the tree is always **logarithmic** with respect to the number of nodes.
- We obtain a **logarithmic cost** for the operations of search, lookup, insertion, and deletion, assuming that the last two are implemented in such a way that they keep the properties of the balanced tree.

AVL trees

- AVL trees are binary search trees satisfying the additional constraint in each node that the **difference between the depth of both children is at most one**.
- Then the **depth** of the tree is always **logarithmic** with respect to the number of nodes.
- We obtain a **logarithmic cost** for the operations of search, lookup, insertion, and deletion, assuming that the last two are implemented in such a way that they keep the properties of the balanced tree.

```
fmod AVL{X :: TOSET<, Y :: CONTENTS} is
  including SEARCH-TREE{X, Y} .
```

```
  --- We add a new field to the node's record.
```

```
  op depth:_ : Nat -> Record [ctor] .
```

```
  op depth : Record -> Nat .
```

```
  var N : Nat .
```

```
  vars Rec Rec' Rec'' : Record .
```

```
  eq depth(Rec, depth: N) = N .
```


AVL trees

```
sorts NeAVL{X, Y} AVL{X, Y} .
subsorts NeAVL{X, Y} < AVL{X, Y} < SearchTree{X, Y} .
subsorts NeAVL{X, Y} < NeSearchTree{X, Y} .
vars L R : AVL{X, Y} .
var ST : NeSearchTree{X, Y} .

mb empty : AVL{X, Y} .
cmb ST : NeAVL{X, Y}
  if L [Rec] R := ST /\ sd(depth(L),depth(R)) <= 1 /\
    1 + max(depth(L),depth(R)) = depth(Rec) .
```

AVL trees

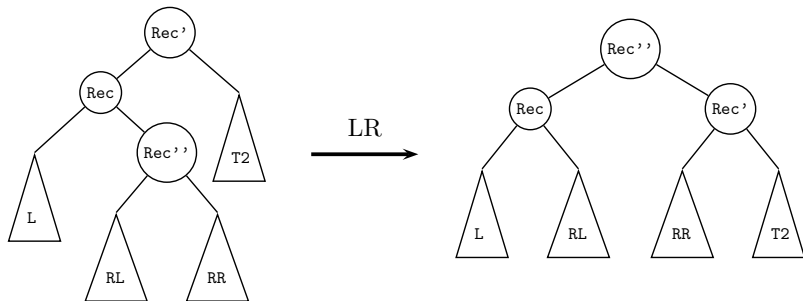
```
op insertAVL : AVL{X, Y} X$Elt Y$Contents -> NeAVL{X, Y} .
op deleteAVL : X$Elt AVL{X, Y} -> AVL{X, Y} .
op depthAVL : AVL{X, Y} -> Nat .
op buildAVL : AVL{X, Y} Record AVL{X, Y} -> AVL{X, Y} .
op join : AVL{X, Y} Record AVL{X, Y} -> AVL{X, Y} .
op lRotate : AVL{X, Y} Record AVL{X, Y} -> AVL{X, Y} .
op rRotate : AVL{X, Y} Record AVL{X, Y} -> AVL{X, Y} .
```

```
vars K K' : X$Elt .
vars C C' : Y$Contents .
```

```
*** EQUATIONS NOT SHOWN
```

```
endfm
```

AVL trees

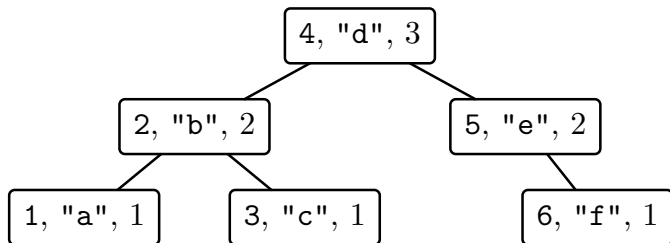


AVL trees

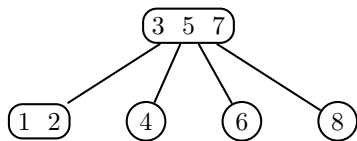
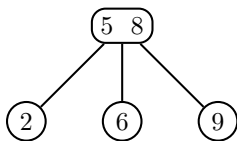
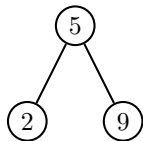
```
fmod AVL-TEST is
  protecting AVL{IntAsToset<, StringAsContents} .
endfm
```

```
Maude> red insertAVL(insertAVL(insertAVL(insertAVL(
      insertAVL(insertAVL(empty, 1, "a"), 2, "b"), 3, "c"),
      4, "d"), 5, "e"), 6, "f") .
result NeAVL{IntAsTOSET<,StringAsContents}:
  ((empty[key: 1,contents: "a",depth: 1]empty)
   [key: 2,contents: "b",depth: 2]
   (empty[key: 3,contents: "c",depth: 1]empty))
 [key: 4,contents: "d",depth:3]
 (empty
  [key: 5,contents: "e",depth: 2]
  (empty[key: 6,contents: "f",depth: 1]empty))
```

AVL trees



2-3-4 trees



2-3-4 trees

```

op empty234 : -> 234Tree{T} [ctor] .
op _[_]_ : 234Tree?{T} T$Elt 234Tree?{T} -> Ne234Tree?{T} [ctor] .
op _<_>_<_>_ : 234Tree?{T} T$Elt 234Tree?{T} T$Elt 234Tree?{T}
    -> Ne234Tree?{T} [ctor] .
op _{ _ }_{ _ }_{ _ }_ : 234Tree?{T} T$Elt 234Tree?{T} T$Elt
    234Tree?{T} T$Elt 234Tree?{T} -> Ne234Tree?{T} [ctor] .

cmb TL [ N ] TR : Ne234Tree{T}
  if greaterKey(N,TL) /\ smallerKey(N,TR) /\
    depth(TL) = depth(TR) .
cmb TL < N1 > TC < N2 > TR : Ne234Tree{T}
  if N1 < N2 /\
    greaterKey(N1,TL) /\ smallerKey(N1,TC) /\
    greaterKey(N2,TC) /\ smallerKey(N2,TR) /\
    depth(TL) = depth(TC) /\ depth(TC) = depth(TR) .
cmb TL { N1 } TLM { N2 } TRM { N3 } TR : Ne234Tree{T}
  if N1 < N2 /\ N2 < N3 /\
    greaterKey(N1,TL) /\ smallerKey(N1,TLM) /\
    greaterKey(N2,TLM) /\ smallerKey(N2,TRM) /\
    greaterKey(N3,TRM) /\ smallerKey(N3,TR) /\
    depth(TL) = depth(TLM) /\ depth(TL) = depth(TRM) /\
    depth(TL) = depth(TR) .

```

2-3-4 trees

```

eq find(M, empty234) = false .
ceq find(M, T1 [N1] T2) = find(M, T1) if M < N1 .
eq find(M, T1 [M] T2) = true .
ceq find(M, T1 [N1] T2) = find(M, T2) if M > N1 .
ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T1) if M < N1 .
eq find(M, T1 < M > T2 < N2 > T3) = true .
ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T2)
  if M > N1 and M < N2 .
eq find(M, T1 < N1 > T2 < M > T3) = true .
ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T3) if M > N2 .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T1)
  if M < N1 .
eq find(M, T1 { M } T2 { N2 } T3 { N3 } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T2)
  if M > N1 and M < N2 .
eq find(M, T1 { N1 } T2 { M } T3 { N3 } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T3)
  if M > N2 and M < N3 .
eq find(M, T1 { N1 } T2 { N2 } T3 { M } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T4)
  if M > N3 .

```


Red-black trees

```

sorts NeRBTree{X, Y} RBTree{X, Y} .
subsort NeRBTree{X, Y} < RBTree{X, Y} < SearchTree{X, Y} .
subsort NeRBTree{X, Y} < NeSearchTree{X, Y} .

```

```

mb empty : RBTree{X, Y} .
cmb ST : NeRBTree{X, Y}
  if RBTL? [Rec] RBTR? := ST /\ color(Rec) = b /\
    blackDepth(RBTR?) = blackDepth(RBTR?) /\
    blackBalance(RBTL?) /\ blackBalance(RBTR?) /\
    not twoRed(RBTL?) /\ not twoRed(RBTR?) .

```

--- Auxiliary operations

```

op blackDepth : BinTree{Record} -> Nat .
op blackBalance : BinTree{Record} -> Bool .
op twoRed : BinTree{Record} -> Bool .

```

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.

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} is a tuple $\mathcal{R} = (\Omega, E, L, R)$ where (Ω, E) is a signature, L is a set of labels, and R is a set of **labelled rewrite rules** written

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

where r is a label and $[t]_E$ and $[t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega, E}(X)$.

- The more general form of a rewrite rule is **conditional**:

$$r : [t]_E \longrightarrow [t']_E \text{ if } \left(\bigwedge_i u_i = v_i \right) \wedge \left(\bigwedge_j w_j : s_j \right) \wedge \left(\bigwedge_k [p_k]_E \longrightarrow [q_k]_E \right)$$

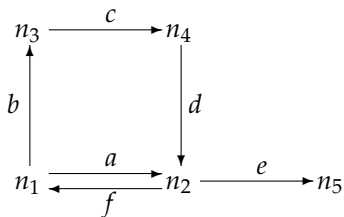
System modules

- **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

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

Transition systems



mod A-TRANSITION-SYSTEM is

sort State .

ops n1 n2 n3 n4 n5 : -> State [ctor] .

r1 [a] : n1 => n2 . r1 [b] : n1 => n3 .

r1 [c] : n3 => n4 . r1 [d] : n4 => n2 .

r1 [e] : n2 => n5 . r1 [f] : n2 => n1 .

endm

- **not** confluent: there are, for example, two transitions out of n_2 that are not joinable
- **not** terminating: there are cycles creating infinite computations

A vending machine

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item State .
  subsorts Coin Item < State .
  op __ : State State -> State [assoc comm] .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
  op c : -> Item [format (b! o)] .
endfm
```

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : State .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```


A vending machine

- The top-down rule-fair **rewrite** command (abbreviated **rew**) applies rules on the top operator (`_` in this case) in a fair way; in this example, it tries to apply `add-q`, `add-$`, and `change` in this order.

```
Maude> rew [20] $ $ q q .
rewrite [20] in VENDING-MACHINE : $ $ q q .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result State: $ $ $ $ $ $ $ $ $ $ $ $ $ $ q q q
```

A vending machine

- The top-down rule-fair **rewrite** command (abbreviated **rew**) applies rules on the top operator (`_` in this case) in a fair way; in this example, it tries to apply `add-q`, `add-$`, and `change` in this order.

```
Maude> rew [20] $ $ q q .
rewrite [20] in VENDING-MACHINE : $ $ q q .
rewrites: 20 in 0ms cpu (0ms real) (~ rewrites/second)
result State: $ $ $ $ $ $ $ $ $ $ $ $ $ q q q
```

- The **frewrite** command (abbreviated **frew**) uses a depth-first position-fair strategy that makes it possible for some rules to be applied that could be “starved” using the `rewrite` command.

```
Maude> frew [20] $ $ q q .
frewrite [20] in VENDING-MACHINE : $ $ q q .
rewrites: 20 in 0ms cpu (1ms real) (~ rewrites/second)
result (sort not calculated):
  c (q a) ($ q) ($ $) c $ q q q q q q q q q a c
```

A vending machine

- With the `search` command we can look for different ways to use a dollar and three quarters to buy an apple and two cakes. First we ask for one solution, and then use the bounded `continue` command to see another solution.

```
Maude> search [1] in VENDING-MACHINE : $ q q q =>+ a c c M .
```

```
Solution 1 (state 108)
```

```
states: 109 rewrites: 1857 in 20ms cpu (57ms real)  
      (92850 rewrites/second)
```

```
M --> q q q q
```

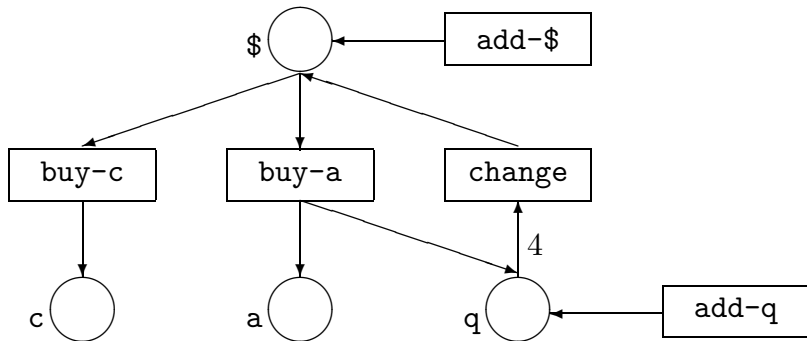
```
Maude> cont 1 .
```

```
Solution 2 (state 160)
```

```
states: 161 rewrites: 1471 in 20ms cpu (76ms real)  
      (73550 rewrites/second)
```

```
M --> q q q q q
```

A vending machine



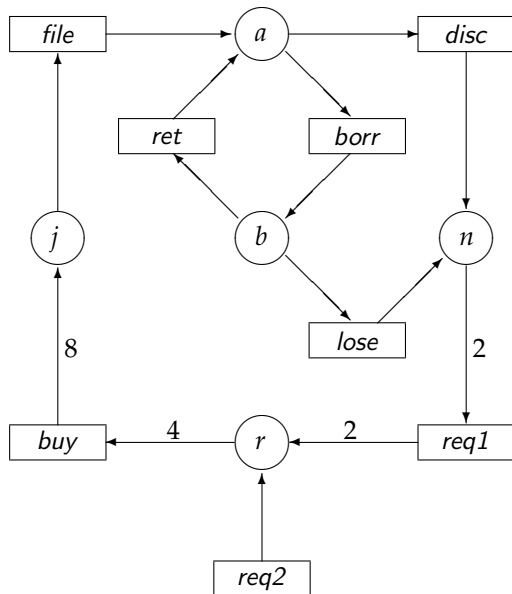
Petri nets

Consider a **Petri net modelling a small library**, where a token represents a book, that can be in several different states: just bought (j), available (a), borrowed (b), requested (r), and not available (n).

The possible transitions are the following:

- *buy*: When there are **four** accumulated requests, the library places an order to buy **two** copies of each requested book (here we do not distinguish among different books or copies of the same book).
- *file*: A book just bought is filed, making it available.
- *borr*: An available book can be borrowed.
- *ret*: A borrowed book can be returned.
- *lose*: A borrowed book can become lost, and thus no longer available.
- *disc*: An available book is discarded because of its bad condition, and thus it is no longer available either.
- *req1*: A user may place a request to buy a non available book, but only when there are **two** accumulated requests these are honored.
- *req2*: The library may decide to buy a **new** book, thus creating a new token in the system.

Petri nets



Petri nets as system modules

- A **marking** on a Petri net is a **multiset** over its set of places, denoting the available resources in each place.
- A **transition** goes from the marking representing the resources it **consumes** to the marking representing the resources it **produces**.

Petri nets as system modules

- A **marking** on a Petri net is a **multiset** over its set of places, denoting the available resources in each place.
- A **transition** goes from the marking representing the resources it **consumes** to the marking representing the resources it **produces**.

```

mod LIBRARY-PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op 1 : -> Marking [ctor] .
  op __ : Marking Marking -> Marking [ctor assoc comm id: 1] .
  ops a b n r j : -> Place [ctor] .
  var M : Marking .
  rl [buy] : r r r r => j j j j j j j j .
  rl [file] : j => a .           rl [borrr] : a => b .
  rl [ret] : b => a .           rl [lose] : b => n .
  rl [disc] : a => n .          rl [req1] : n n => r r .
  rl [req2] : M => M r .
endm

```


Petri nets as system modules

- Computations may be **nonterminating**, for example, because a book can be forever in the borrow-return cycle.
- This particular net happens to be confluent by chance.

Petri nets as system modules

- Computations may be **nonterminating**, for example, because a book can be forever in the borrow-return cycle.
- This particular net happens to be confluent by chance.
- Starting in the **empty marking** and using the **rewrite** command, the system keeps adding requests until there are enough to buy a book, and then repeats the same process, like in the following sequence of 10 rewrites.

```
Maude> rew [10] 1 .  
rewrite [10] in LIBRARY-PETRI-NET : 1 .  
rewrites: 10 in 20ms cpu (54ms real) (500 rewrites/second)  
result Marking: j j j j j j j j j j j j j j j
```

Petri nets as system modules

- Computations may be **nonterminating**, for example, because a book can be forever in the borrow-return cycle.
- This particular net happens to be confluent by chance.
- Starting in the **empty marking** and using the **rewrite** command, the system keeps adding requests until there are enough to buy a book, and then repeats the same process, like in the following sequence of 10 rewrites.

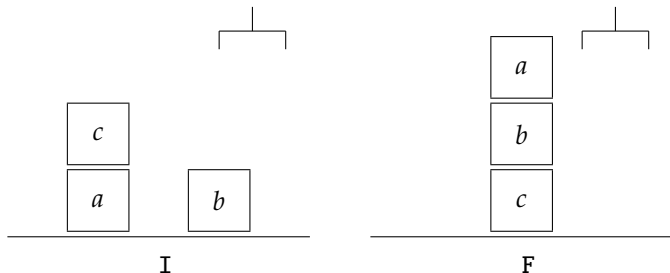
```
Maude> rew [10] 1 .
rewrite [10] in LIBRARY-PETRI-NET : 1 .
rewrites: 10 in 20ms cpu (54ms real) (500 rewrites/second)
result Marking: j j j j j j j j j j j j j j j j
```

- The **frewrite** command uses a **fair** strategy where other rules are applied.

```
Maude> frewrite [10] 1 .
frewrite [10] in LIBRARY-PETRI-NET : 1 .
rewrites: 10 in 0ms cpu (19ms real) (~ rewrites/second)
result (sort not calculated): a (r j) a (r j) a j j j
```

Blocks world

- A generalization of Petri net computations is provided by **conjunctive planning** problems, where the states are described by means of some kind of conjunction of propositions describing basic facts.
- A typical example is the **blocks world**; here we have a table on top of which there are blocks, which can be moved only by means of a robot arm.



Blocks world

```

mod BLOCKS-WORLD is
  protecting QID .
  sorts BlockId Prop State .
  subsort Qid < BlockId .
  subsort Prop < State .

  op table : BlockId -> Prop [ctor] .      *** block is on the table
  op on : BlockId BlockId -> Prop [ctor] . *** block A is on block B
  op clear : BlockId -> Prop [ctor] .      *** block is clear
  op hold : BlockId -> Prop [ctor] .       *** robot arm holds block
  op empty : -> Prop [ctor] .              *** robot arm is empty

  op 1 : -> State [ctor] .
  op _&_ : State State -> State [ctor assoc comm id: 1] .

```

Blocks world

```
vars X Y : BlockId .

rl [pickup] : empty & clear(X) & table(X) => hold(X) .
rl [putdown] : hold(X) => empty & clear(X) & table(X) .
rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
endm
```

Blocks world

```

vars X Y : BlockId .

rl [pickup] : empty & clear(X) & table(X) => hold(X) .
rl [putdown] : hold(X) => empty & clear(X) & table(X) .
rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
endm

```

If we just ask for a sequence of **rewrites** starting from the initial state I , it gets very boring as the robot arm, after picking up the block b keeps stacking and unstacking it on c ; we show a sequence of 5 rewrites:

```

Maude> rew [5] empty & clear('c) & clear('b) & table('a) &
          table('b) & on('c, 'a) .
rewrite [5] in BLOCKS-WORLD : empty & clear('c) & clear('b) &
          table('a) & table('b) & on('c, 'a) .
rewrites: 5 in 0ms cpu (42ms real) (~ rewrites/second)
result State: table('a) & clear('c) & hold('b) & on('c, 'a)

```

Blocks world

To see that it is possible to go from state I to state F we can use the `search` command as follows:

```
Maude> search empty & clear('c) & clear('b) & table('a) &
        table('b) & on('c,'a)
        =>* empty & clear('a) & table('c) & on('a,'b) & on('b,'c) .
```

Solution 1 (state 21)

empty substitution

No more solutions.

```
Maude> show path labels 21 .
```

```
unstack
putdown
pickup
stack
pickup
stack
```


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 Josephus problem

- Flavius Josephus was a famous Jewish historian who, during the Jewish-Roman war in the first century, was trapped in a cave with a group of 40 Jewish soldiers surrounded by Romans.
- Legend has it that, preferring death to being captured, the Jews decided to gather in a **circle** and rotate a **dagger** around so that every **third** remaining person would commit suicide.
- Apparently, Josephus was too keen to live and quickly found out the **safe** position.

The Josephus problem

- Flavius Josephus was a famous Jewish historian who, during the Jewish-Roman war in the first century, was trapped in a cave with a group of 40 Jewish soldiers surrounded by Romans.
- Legend has it that, preferring death to being captured, the Jews decided to gather in a **circle** and rotate a **dagger** around so that every **third** remaining person would commit suicide.
- Apparently, Josephus was too keen to live and quickly found out the **safe** position.
- The circle representation becomes a (circular) **list** once the beginning position is chosen, with the dagger implicitly at position 1.
- The idea then consists in continually taking the first two elements in the list and moving them to the end of it while “killing” the third one.
- The dagger remains always **implicitly** located at the beginning of the list.

The Josephus problem

```

mod JOSEPHUS is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .

  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem -> Circle .
  op initial : NzNat -> Moriturem .

  var M : Moriturem .
  vars I1 I2 I3 N : NzNat .

  eq initial(1) = 1 .
  eq initial(s(N)) = initial(N) s(N) .

  rl [kill>3] : { I1 I2 I3 M } => { M I1 I2 } .
  rl [kill3] : { I1 I2 I3 } => { I1 I2 } .
  --- Rule kill3 is necessary because M cannot be empty
  rl [kill2] : { I1 I2 } => { I2 } .
endm

```

The Josephus problem

- In this problem, transitions are **deterministic**: only one of the three can be applied each time, depending on the numbers of elements remaining in the list.
- Search can be used, but it is not necessary.
- The command **rewrite** provides directly the solution:

```
Maude> rewrite { initial(41) } .  
result Circle: {31}
```


The generalized Josephus problem

- It is easy to generalize the program so that every *i*-th person commits suicide, where *i* is a parameter.
- The idea is the same, but because of the parameter now it is necessary to **explicitly** represent the dagger.
- For that, we use the constructor

```
dagger : NzNat NzNat -> Moriturem .
```

whose second argument stores the value of *i* while the first one acts as a counter.

- Each time an element is moved from the beginning of the list to the end, the first argument is decreased by one; once it reaches 1, the element that is currently the head of the list is “killed,” i.e., removed from the list.

The generalized Josephus problem

```

mod JOSEPHUS-GENERALIZED is
  protecting NAT .
  sorts Moriturem Circle .
  subsort NzNat < Moriturem .
  op dagger : NzNat NzNat -> Moriturem .
  op __ : Moriturem Moriturem -> Moriturem [assoc] .
  op {_} : Moriturem -> Circle .
  op initial : NzNat NzNat -> Moriturem .

  var M : Moriturem .
  vars I I1 I2 N : NzNat .
  eq initial(1, I) = dagger(I, I) 1 .
  eq initial(s(N), I) = initial(N, I) s(N) .

  rl [kill] : { dagger(1, I) I1 M } => { dagger(I, I) M } .
  rl [next] : { dagger(s(N), I) I1 M } => { dagger(N, I) M I1 } .
  rl [last] : { dagger(N, I) I1 } => { I1 } .
  --- The last one throws the dagger away!

endm

```

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.

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 **singers**, the **flashlight**, and a **watch** to keep record of the time.
- The flashlight and the singers have a **Place** associated to them, indicating whether their current position is to the left or to the right of the bridge.
- Each singer, 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 Singer Object Group Place .
  subsorts Singer Object < Group .

  ops left right : -> Place .
  op flashlight : Place -> Object .
  op watch : Nat -> Object .
  op singer : Nat Place -> Singer .
  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) singer(1, left)
    singer(2, left) singer(5, left) singer(10, left) .

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

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

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

endm

```

Crossing the bridge

- A solution can be found by looking for a state in which all singers 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)
    singer(1, right) singer(2, right)
    singer(5, right) singer(10, right)
    such that N:Nat <= 17 .
```

Solution 1 (state 402)

N --> 17

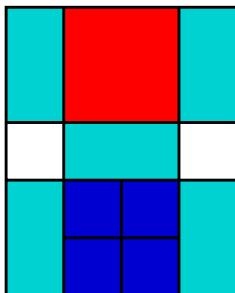
Crossing the bridge

- 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.

Black or white

- In an 8×8 board the **four corners** are colored **white** and all the **others are black**.
- Is it possible to make **all the squares white** by recoloring rows and columns?
- **Recoloring** is the operation of changing the colors of all the squares in a **row** or **column**.

Black or white

- In an 8×8 board the **four corners** are colored **white** and all the **others are black**.
- Is it possible to make **all the squares white** by recoloring rows and columns?
- **Recoloring** is the operation of changing the colors of all the squares in a **row** or **column**.
- The board is represented as a **multiset** of squares.
- Each square has three arguments: the first two are its **column** and **row**, and the last one its **color**, either black (b) or white (w).
- A predicate **allWhite?** checks whether all the squares are white.

Black or white

```
mod RECOLORING is
  protecting NAT .
  sorts Place Board Color .
  subsort Place < Board .
  ops w b : -> Color .
  op sq : Nat Nat Color -> Place .
  op __ : Board Board -> Board [assoc comm] .

  op recolor : Color -> Color .
  op allWhite? : Board -> Bool .

  vars I J : Nat .
  vars C1 C2 C3 C4 C5 C6 C7 C8 : Color .
  var B : Board .

  eq recolor(b) = w .
  eq recolor(w) = b .

  eq allWhite?(sq(I,J,b) B) = false .
  eq allWhite?(B) = true [owise] .
```

Black or white

```
op initial : -> Board .
```

```
eq initial = sq(1,1,b) sq(1,2,w) ..... sq(8,7,w) sq(8,8,b) .
```

```
rl [recolor-column] : sq(I,1,C1) sq(I,2,C2) sq(I,3,C3) sq(I,4,C4)
    sq(I,5,C5) sq(I,6,C6) sq(I,7,C7) sq(I,8,C8)
=> sq(I,1,recolor(C1)) sq(I,2,recolor(C2))
    sq(I,3,recolor(C3)) sq(I,4,recolor(C4))
    sq(I,5,recolor(C5)) sq(I,6,recolor(C6))
    sq(I,7,recolor(C7)) sq(I,8,recolor(C8)) .
```

```
rl [recolor-row] : sq(1,J,C1) sq(2,J,C2) sq(3,J,C3) sq(4,J,C4)
    sq(5,J,C5) sq(6,J,C6) sq(7,J,C7) sq(8,J,C8)
=> sq(1,J,recolor(C1)) sq(2,J,recolor(C2))
    sq(3,J,recolor(C3)) sq(4,J,recolor(C4))
    sq(5,J,recolor(C5)) sq(6,J,recolor(C6))
    sq(7,J,recolor(C7)) sq(8,J,recolor(C8)) .
```

```
endm
```

Black or white

The command

```
Maude> search initial =>* B:Board  
      such that allWhite?(B:Board) .
```

No solution.

proves that it is **not** possible to get a configuration with all the squares colored white from the initial configuration.

The game is not over

- Mr. Smith and his wife invited four other couples to have dinner at their house. When they arrived, some people shook hands with some others (of course, nobody shook hands with their spouse or with the same person twice), after which Mr. Smith asked everyone how many times they had shaken hands. The answers, it turned out, were different in all cases. How many people did Mrs. Smith shake hands with?

The game is not over

- The numbers 25 and 36 are written on a blackboard. At each turn, a player writes on the board the (positive) difference between two numbers already on the blackboard—if this number does not already appear on it. The loser is the player who cannot write a number. Prove that the second player will always win.
- A 3×3 table is filled with numbers. It is allowed to increase simultaneously all the numbers in any 2×2 square by 1. Is it possible, using these operations, to obtain the table $(4, 9, 5), (10, 18, 12), (6, 13, 7)$ from a table initially filled with zeros?

Untyped lambda calculus

```

fth VAR is
  protecting BOOL .
  sorts Var VarSet .
  subsort Var < VarSet .
  op empty-set : -> VarSet .
  op _U_ : VarSet VarSet -> VarSet [assoc comm id: empty-set] .
  op _in_ : Var VarSet -> Bool .
  op _\_ : VarSet VarSet -> VarSet .
  op new : VarSet -> Var .

  vars E E' : Var .
  vars S S' : VarSet .

  eq E U E = E .
  eq E in empty-set = false .
  eq E in E' U S = (E == E') or (E in S) .
  eq empty-set \ S = empty-set .
  eq (E U S) \ S' = if E in S' then S \ S' else E U (S \ S') fi .
  eq new(S) in S = false [nonexec] .
endfth

```

Untyped lambda calculus

```

fmod LAMBDA{X :: VAR} is
  sort Lambda{X} .
  subsort X$Var < Lambda{X} .          *** variables
  op \_._ : X$Var Lambda{X} -> Lambda{X} [ctor] .
                                          *** lambda abstraction
  op __ : Lambda{X} Lambda{X} -> Lambda{X} [ctor] .
                                          *** application
  op _[_/_] : Lambda{X} Lambda{X} X$Var -> Lambda{X} .
                                          *** substitution
  op fv : Lambda{X} -> X$VarSet .      *** free variables

vars X Y : X$Var .
vars M N P : Lambda{X} .

*** Free variables
eq fv(X) = X .
eq fv(\ X . M) = fv(M) \ X .
eq fv(M N) = fv(M) U fv(N) .
eq fv(M [N / X]) = (fv(M) \ X) U fv(N) .

```

Untyped lambda calculus

```

*** Substitution equations
eq X [N / X] = N .
ceq Y [N / X] = Y if X /= Y .
eq (M N)[P / X] = (M [P / X])(N [P / X]) .
eq (\ X . M)[N / X] = \ X . M .
ceq (\ Y . M)[N / X] = \ Y . (M [N / X])
  if X /= Y and (not(Y in fv(N)) or not(X in fv(M))) .
ceq (\ Y . M)[N / X]
  = \ (new(fv(M N))) . ((M [new(fv(M N)) / Y])[N / X])
  if X /= Y /\ (Y in fv(N)) /\ (X in fv(M)) .

*** Alpha conversion
ceq \ X . M = \ Y . (M [Y / X]) if not(Y in fv(M)) [nonexec] .
endfm

```

Untyped lambda calculus

```
mod BETA-ETA{X :: VAR} is
  including LAMBDA{X} .
  var X : X$Var .
  vars M N : Lambda{X} .
  rl [beta] : (\ X . M) N => M [N / X] .
  crl [eta] : \ X . (M X) => M if not(X in fv(M)) .
endm
```

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

Untyped lambda calculus

```
fmod NAT-SET-MAX is
  protecting (SET * (op _,_ to _U_)){Nat} .
  op max : Set{Nat} -> Nat .
  var N : Nat .
  var S : Set{Nat} .
  eq max(empty) = 0 .
  eq max(N U S) = if N > max(S) then N else max(S) fi .
endfm
```

```
view VarNat from VAR to NAT-SET-MAX is
  sort Var to Nat .
  sort VarSet to Set{Nat} .
  var S : VarSet .
  op empty-set to empty .
  op new(S) to term max(S) + 1 .
endv
```

```
mod UNTYPED-LAMBDA-CALCULUS is
  protecting BETA-ETA{VarNat} .
endm
```

Untyped lambda calculus

```

Maude> set trace on .
Maude> set trace eqs off .
Maude> rew [2] (\ 1 . (1 1))(\ 1 . (1 1)) .
rewrite [2] in UNTYPED-LAMBDA-CALCULUS : (\ 1 . (1 1)) \ 1 . (1 1) .
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> 1 1
N:Lambda{VarNat} --> \ 1 . (1 1)
(\ 1 . (1 1)) \ 1 . (1 1)
--->
(1 1)[\ 1 . (1 1) / 1]
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> 1 1
N:Lambda{VarNat} --> \ 1 . (1 1)
(\ 1 . (1 1)) \ 1 . (1 1)
--->
(1 1)[\ 1 . (1 1) / 1]
rewrites: 8 in 10ms cpu (31ms real) (800 rewrites/second)
result Lambda{VarNat}: (\ 1 . (1 1)) \ 1 . (1 1)

```


Untyped lambda calculus

```

Maude> set trace on .
Maude> set trace eqs off .
Maude> rew (\ 1 . (\ 2 . 2))((\ 1 . (1 1))(\ 1 .(1 1))) .

rewrite in UNTYPED-LAMBDA-CALCULUS :
  (\ 1 . \ 2 . 2) ((\ 1 . (1 1)) \ 1 . (1 1)) .
***** rule
r1 (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> \ 2 . 2
N:Lambda{VarNat} --> (\ 1 . (1 1)) \ 1 . (1 1)
(\ 1 . \ 2 . 2) ((\ 1 . (1 1)) \ 1 . (1 1))
--->
(\ 2 . 2)[(\ 1 . (1 1)) \ 1 . (1 1) / 1]
rewrites: 36 in 0ms cpu (0ms real) (~ rewrites/second)
result Lambda{VarNat}: \ 2 . 2

```

Object-oriented systems

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

$$\langle 0 : C \mid a_1 : v_1, \dots, a_n : v_n \rangle$$

where 0 is the object's **name**, belonging to a set $0id$ of object identifiers, C is its **class**, the a_i 's are the names of the object's **attributes**, and the v_i '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.

Object-oriented rules

- General form of **rules in object-oriented systems**:

$$\begin{aligned}
 &M_1 \dots M_n \langle O_1 : F_1 \mid \text{atts}_1 \rangle \dots \langle O_m : F_m \mid \text{atts}_m \rangle \\
 &\longrightarrow \langle O_{i_1} : F'_{i_1} \mid \text{atts}'_{i_1} \rangle \dots \langle O_{i_k} : F'_{i_k} \mid \text{atts}'_{i_k} \rangle \\
 &\quad \langle Q_1 : D_1 \mid \text{atts}''_1 \rangle \dots \langle Q_p : D_p \mid \text{atts}''_p \rangle \\
 &\quad M'_1 \dots M'_q \\
 &\text{if } C
 \end{aligned}$$

- By convention, the only object attributes made **explicit** in a rule are those **relevant** for that rule:
 - the attributes mentioned only in the lefthand side of the rule are preserved unchanged,
 - the original values of attributes mentioned only in the righthand side of the rule do not matter, and
 - all attributes not explicitly mentioned are left unchanged.

Another puzzle

7	1	2
6		8
5	4	3

7	8	1
6		2
5	4	3

Another puzzle

7	1	2
6		8
5	4	3

7	8	1
6		2
5	4	3

eq **initial**

```
= < (1, 1) : Tile | value : 7 >    < (1, 2) : Tile | value : 1 >
   < (1, 3) : Tile | value : 2 >    < (2, 1) : Tile | value : 6 >
   < (2, 2) : Tile | value : blank > < (2, 3) : Tile | value : 8 >
   < (3, 1) : Tile | value : 5 >    < (3, 2) : Tile | value : 4 >
   < (3, 3) : Tile | value : 3 > .
```

eq **final**

```
= < (1, 1) : Tile | value : 7 >    < (1, 2) : Tile | value : 8 >
   < (1, 3) : Tile | value : 1 >    < (2, 1) : Tile | value : 6 >
   < (2, 2) : Tile | value : blank > < (2, 3) : Tile | value : 2 >
   < (3, 1) : Tile | value : 5 >    < (3, 2) : Tile | value : 4 >
   < (3, 3) : Tile | value : 3 > .
```

Another puzzle

(omod 8-PUZZLE is

```

  sorts NumValue Value Coordinate .
  subsort NumValue < Value .
  ops 1 2 3 4 : -> Coordinate [ctor] .
  ops 1 2 3 4 5 6 7 8 : -> NumValue [ctor] .
  op blank : -> Value [ctor] .

```

```

  op s_ : Coordinate -> Coordinate .
  eq s 1 = 2 .
  eq s 2 = 3 .
  eq s 3 = 4 .
  eq s 4 = 4 .

```

```

  op '(_',_') : Coordinate Coordinate -> Oid .
  class Tile | value : Value .
  msgs left right up down : -> Msg .

```

```

  vars N M R : Coordinate .
  var P : NumValue .

```

Another puzzle

```
crl [r] : right
  < (N, R) : Tile | value : blank > < (N, M) : Tile | value : P >
=> < (N, R) : Tile | value : P > < (N, M) : Tile | value : blank >
if R == s M .
```

```
crl [l] : left
  < (N, R) : Tile | value : blank > < (N, M) : Tile | value : P >
=> < (N, R) : Tile | value : P > < (N, M) : Tile | value : blank >
if s R == M .
```

```
crl [u] : up
  < (R, M) : Tile | value : blank > < (N, M) : Tile | value : P >
=> < (R, M) : Tile | value : P > < (N, M) : Tile | value : blank >
if s R == N .
```

```
crl [d] : down
  < (R, M) : Tile | value : blank > < (N, M) : Tile | value : P >
=> < (R, M) : Tile | value : P > < (N, M) : Tile | value : blank >
if R == s N .
```

```
ops initial final : -> Configuration .
```

```
eq initial = ... .
```

```
eq final = ... .
```

```
endom)
```

Another puzzle

```
Maude > (search up left down right initial =>* C:Configuration
        such that C:Configuration == final .)
rewrites: 974 in 50ms cpu (68ms real) (19480 rewrites/second)
```

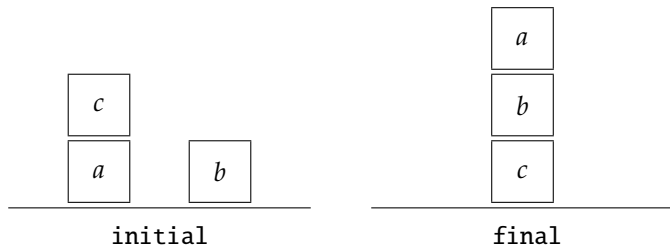
Solution 1

```
C:Configuration -->
```

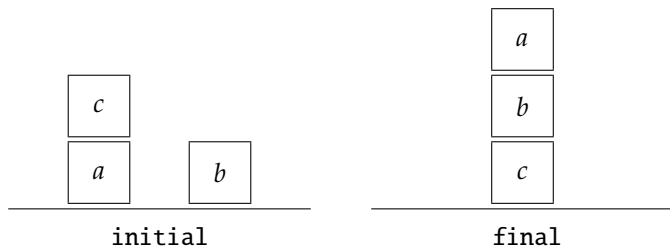
```
<(1,1): Tile | value : 7 > <(1,2): Tile | value : 8 >
<(1,3): Tile | value : 1 > <(2,1): Tile | value : 6 >
<(2,2): Tile | value : blank > <(2,3): Tile | value : 2 >
<(3,1): Tile | value : 5 > <(3,2): Tile | value : 4 >
<(3,3): Tile | value : 3 >
```

No more solutions.

An object-oriented blocks world



An object-oriented blocks world



eq **initial**

```
= < 'a : Block | under : 'c, on : table >
  < 'c : Block | under : clear, on : 'a >
  < 'b : Block | under : clear, on : table > .
```

eq **final**

```
= < 'a : Block | under : clear, on : 'b >
  < 'b : Block | under : 'a, on : 'c >
  < 'c : Block | under : 'b, on : table > .
```

An object-oriented blocks world

```
(omod OO-BLOCKS-WORLD is
  protecting QID .

  sorts BlockId Up Down .
  subsorts Qid < BlockId < Oid .
  subsorts BlockId < Up Down .

  op clear : -> Up [ctor] .
  op table : -> Down [ctor] .

  class Block | under : Up, on : Down .
  msg move : Oid Oid Oid -> Msg .
  msgs unstack stack : Oid Oid -> Msg .

  vars X Y Z : BlockId .
```

An object-oriented blocks world

```

rl [move] : move(X, Z, Y)
  < X : Block | under : clear, on : Z >
  < Z : Block | under : X > < Y : Block | under : clear >
=> < X : Block | on : Y >
    < Z : Block | under : clear > < Y : Block | under : X > .

```

```

rl [unstack] : unstack(X,Z)
  < X : Block | under : clear, on : Z >
  < Z : Block | under : X >
=> < X : Block | on : table > < Z : Block | under : clear > .

```

```

rl [stack] : stack(X, Z)
  < X : Block | under : clear, on : table >
  < Z : Block | under : clear >
=> < X : Block | on : Z > < Z : Block | under : X > .

```

```
ops initial final : -> Configuration .
```

```
eq initial = ... .
```

```
eq final = ... .
```

```
endom)
```

An object-oriented blocks world

```
Maude > (search unstack('c,'a) stack('b,'c) stack('a,'b) initial
=>* C:Configuration such that C:Configuration == final .)
rewrites: 1318 in 20ms cpu (107ms real) (65900 rewrites/second)
```

Solution 1

```
C:Configuration --> < 'a : Block | on : 'b, under : clear >
                   < 'b : Block | on : 'c, under : 'a >
                   < 'c : Block | on : table, under : 'b >
```

No more solutions.

Colored blocks

```
(omod OO-BLOCKS-WORLD+COLOR is
  including OO-BLOCKS-WORLD .
  sort Color .
  ops red blue yellow : -> Color [ctor] .
  class ColoredBlock | color : Color .
  subclass ColoredBlock < Block .
endom)
```

Colored blocks

```
(omod OO-BLOCKS-WORLD+COLOR is
  including OO-BLOCKS-WORLD .
  sort Color .
  ops red blue yellow : -> Color [ctor] .
  class ColoredBlock | color : Color .
  subclass ColoredBlock < Block .
endom)
```

```
Maude> (rew unstack('c, 'a)
  < 'a : Block | color : red, under : 'c, on : table >
  < 'c : Block | color : blue, under : clear, on : 'a >
  < 'b : Block | color : yellow, under : clear, on : table > .)
```

Result Configuration :

```
< 'a : Block | on : table , under : clear , color : red >
< 'c : Block | on : table , under : clear , color : blue >
< 'b : Block | on : table , under : clear , color : yellow >
```

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.

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

Linear temporal logic

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

Kripke structures

- A **Kripke structure** is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that
 - A is a set, called the set of **states**,
 - $\rightarrow_{\mathcal{A}}$ is a total binary relation on A , called the **transition relation**, and
 - $L : A \rightarrow \mathcal{P}(AP)$ 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 .
- The semantics of the temporal logic LTL is defined by means of a **satisfaction relation**

$$\mathcal{A}, a \models \varphi$$

between a Kripke structure \mathcal{A} having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in \text{LTL}(AP)$.

Kripke structures

- Given a system module \mathbb{M} specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, we
 - choose a kind k in \mathbb{M} as our **kind of states**;
 - define some **state predicates** Π and their semantics in a module, say \mathbb{M} -PREDS, protecting \mathbb{M} by means of the (partial) operation $_|_ = _ : \text{State Formula} \rightarrow \text{Bool}$.
coming from the predefined SATISFACTION module.
- Then we get a Kripke structure

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

- Under some assumptions on \mathbb{M} and \mathbb{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.

Mutual exclusion: processes

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

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

  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 .

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


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

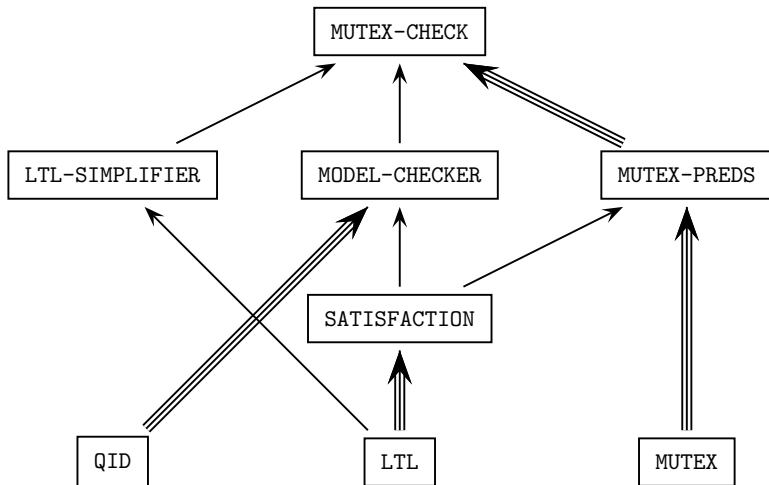
- A **counterexample** is a pair consisting of two lists of transitions, where the first corresponds to a finite path beginning in the initial state, and the second describes a loop.
- If we check whether, beginning in the state `initial1`, process `b` will always be waiting, we get a counterexample:

```
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})
```

Model-checking modules



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.

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 .
  op change : Side -> Side .

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

  var S : Side .

  eq change(left) = right .
  eq change(right) = left .

  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:
$$\langle \diamond \text{ success} \rightarrow (\langle \diamond \text{ disaster} / ((\sim \text{ success}) \cup \text{ disaster})) \rangle$$
- 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

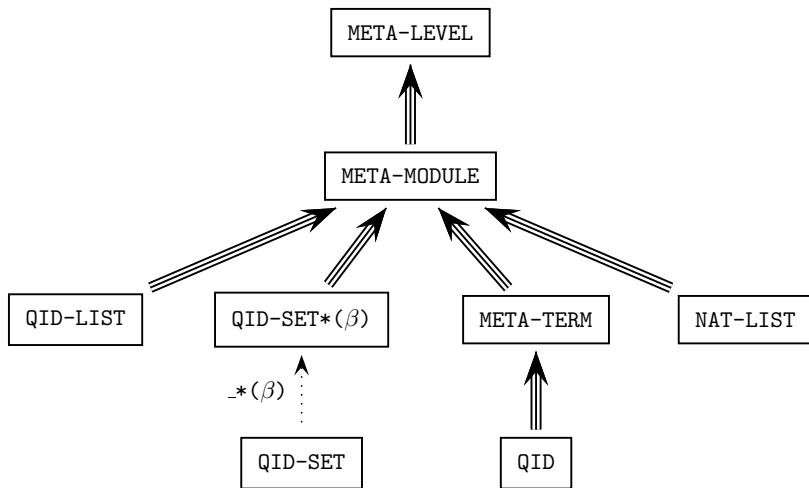
- Rewriting logic is **reflective**, because there is a finitely presented rewrite theory \mathcal{U} that is **universal** in the sense that:
 - we can represent any finitely presented rewrite theory \mathcal{R} and any terms t, t' in \mathcal{R} as **terms** $\overline{\mathcal{R}}$ and $\overline{t},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

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

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.

Representing terms

- Usual **term**: `c (q M:State)`
- **Meta**representation
`'__['c.Item, '__['q.Coin, 'M:State]]`

Representing terms

- Usual **term**: `c (q M:State)`

- **Meta**representation

```
'__['c.Item, '__['q.Coin, 'M:State]]
```

- **Meta-meta**representation

```
'_['['_'] ['___.Qid,
      '_',_[''c.Item.Constant,
            '_['['_'] ['___.Qid,
                  '_',_[''q.Coin.Constant,
                        ''M:State.Variable]]]]]
```

Representing modules

```
fmod 'VENDING-MACHINE-SIGNATURE is
  nil
  sorts 'Coin ; 'Item ; 'State .
  subsort 'Coin < 'State .
  subsort 'Item < 'State .
  op '__ : 'State 'State -> 'State [assoc comm] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  none
  none
endfm
```

Representing modules

```
mod 'VENDING-MACHINE is
  including 'VENDING-MACHINE-SIGNATURE .
  sorts none .
  none
  none
  none
  none
  rl 'M:State => '__[ 'M:State, 'q.Coin ] [label('add-q)] .
  rl 'M:State => '__[ 'M:State, '$.Coin ] [label('add-$)] .
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__[ 'a.Item, 'q.Coin ] [label('buy-a)] .
  rl '__[ 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin ]
    => '$.Coin [label('change)] .
endm
```

metaRewrite

- Its first two arguments are the representations in META-LEVEL of a **module** \mathcal{R} and of a **term** t , and its third argument is a **natural** n .
- Its result is the representation of the **term obtained from t after at most n applications of the rules in \mathcal{R}** using the strategy of Maude's command `rewrite`, together with the metarepresentation of its corresponding sort or kind.

```
Maude> reduce in META-LEVEL :
      metaRewrite(upModule('VENDING-MACHINE, false),
        '___['$.Coin, '___['$.Coin, '___['q.Coin, 'q.Coin]]], 1) .
result ResultPair:
  {'___['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin], 'State}
```

```
Maude> reduce in META-LEVEL :
      metaRewrite(upModule('VENDING-MACHINE, false),
        '___['$.Coin, '___['$.Coin, '___['q.Coin, 'q.Coin]]], 2) .
result ResultPair:
  {'___['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin], 'State}
```

metaApply

- The first three arguments are representations in META-LEVEL of a **module** \mathcal{R} , a **term** t in \mathcal{R} , and a **label** l of some rules in \mathcal{R} .
- The fourth argument is a set of assignments (possibly empty) defining a partial **substitution** σ for the variables in the rules.
- The last argument is a **natural number** n , used to **enumerate** (starting from 0) all the **possible solutions** of the intended rule application.
- It returns a triple of sort `ResultTriple` consisting of the metarepresentation of a term, the metarepresentation of its corresponding sort or kind, and the metarepresentation of a substitution.

metaApply

The operation `meta-apply` is evaluated as follows:

- ① the term t is first **fully reduced** using the equations in \mathcal{R} ;
- ② the resulting term is **matched at the top** against all rules with label l in \mathcal{R} partially instantiated with σ , with matches that fail to satisfy the condition of their rule discarded;
- ③ the **first n successful matches are discarded**; if there is an $(n + 1)$ th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise **failure** is returned;
- ④ the term resulting from **applying the given rule with the $(n + 1)$ th match** is fully reduced using the equations in \mathcal{R} ;
- ⑤ the triple formed by the metarepresentation of the resulting fully reduced term, the metarepresentation of its corresponding sort or kind, and the metarepresentation of the substitution used in the reduction is returned.

metaApply

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '$.Coin, 'buy-c, none, 0) .
result ResultTriple: {'c.Item, 'Item, (none).Substitution}
```

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '$.Coin, 'add-$, none, 0) .
result ResultTriple:
  {'__['$.Coin, '$.Coin], 'State, 'M:State <- '$.Coin}
```

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '__['q.Coin, '$.Coin], 'buy-c, none, 0) .
result ResultTriple?: (failure).ResultTriple?
```

metaSearch

- `metaSearch` takes as arguments
 - the metarepresentation of a `module`,
 - the metarepresentation of the starting `term` for search,
 - the metarepresentation of the `pattern` to search for,
 - the metarepresentation of a `condition` to be satisfied,
 - the metarepresentation of the `type of search` to carry on,
 - a `Bound` value, and
 - a `natural number`, to enumerate solutions.
- The searching strategy used by `metaSearch` coincides with that of the object level `search` command.
- The possible types of search are:
 - `'*` for a search involving zero or more rewrites (corresponding to `=>*` in the `search` command),
 - `'+` for a search consisting in one or more rewrites (`=>+`),
 - `'!` for a search that only matches canonical forms (`=>!`).
- The result has the same form as the result of `metaApply`.

metaSearch

```
Maude> reduce in META-LEVEL :
  metaSearch(upModule('VENDING-MACHINE, false),
    ___['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
    ___['c.Item, 'a.Item, 'c.Item, 'M:State],
    nil, '+, unbounded, 0) .
result ResultTriple:
  {'___['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'c.Item, 'c.Item],
   'State,
   'M:State <- '___['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin]}
```

```
Maude> reduce in META-LEVEL :
  metaSearch(upModule('VENDING-MACHINE, false),
    ___['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
    ___['c.Item, 'a.Item, 'c.Item, 'M:State],
    nil, '+, unbounded, 1) .
result ResultTriple:
  {'___['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'c.Item, 'c.Item],
   'State,
   'M:State <- '___['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin]}
```

metaSearchPath

```
Maude> reduce in META-LEVEL :
    metaSearchPath(upModule('VENDING-MACHINE, false),
        '__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
        '__['c.Item, 'a.Item, 'c.Item, 'M:State],
        nil, '+, unbounded, 0) .
```

result Trace:

```
{'__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
  'State,
  rl 'M:State => '__['$.Coin, 'M:State] [label('add-)] .}
{'__['$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
  'State,
  rl 'M:State => '__['$.Coin, 'M:State] [label('add-)] .}
{'__['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
  'State,
  rl '$.Coin => 'c.Item [label('buy-c)] .}
{'__['$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'c.Item],
  'State,
  rl '$.Coin => 'c.Item [label('buy-c)] .}
{'__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'c.Item, 'c.Item],
  'State,
  rl '$.Coin => '__['q.Coin, 'a.Item] [label('buy-a)] .}
```

Internal strategies

- System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating.
- Therefore, we need to have good ways of **controlling** the rewriting inference process—which in principle could not terminate or go in many undesired directions—by means of adequate **strategies**.
- In Maude, thanks to its reflective capabilities, strategies can be made **internal** to the system.
- That is, they can be defined using statements in a normal module in Maude, and can be reasoned about as with statements in any other module.
- In general, internal strategies are defined in extensions of the META-LEVEL module by using `metaReduce`, `metaApply`, `metaXApply`, etc., as building blocks.

Internal strategies

We illustrate some of these possibilities by implementing the following strategies for controlling the execution of the rules in the VENDING-MACHINE module:

- **insertCoin**: insert either a dollar or a quarter in the vending machine;
- **onlyCakes**: only buy cakes, and buy as many cakes as possible, with the coins already inserted;
- **onlyNitems**: only buy either cakes or apples, and buy at most n of them, with the coins already inserted;
- **cakesAndApples**: buy the same number of apples and cakes, and buy as many as possible, with the coins already inserted.

Internal strategies: insertCoin

```
var T : Term .
var Q : Qid .
var N : Nat .
vars BuyCake? Change? BuyItem? : Result4Tuple? .

op insertCoin : Qid Term -> Term .

eq insertCoin('q.Coin, T)
  = getTerm(metaXapply(upModule('VENDING-MACHINE, false),
    T, 'add-q, none, 0, unbounded, 0)) .
eq insertCoin('$.Coin, T)
  = getTerm(metaXapply(upModule('VENDING-MACHINE, false),
    T, 'add-$, none, 0, unbounded, 0)) .
eq insertCoin(Q, T) = T [owise] .
```

Internal strategies: onlyCakes

```
op onlyCakes : Term -> Term .

ceq onlyCakes(T)
  = if BuyCake? :: Result4Tuple
    then onlyCakes(getTerm(BuyCake?))
    else (if Change? :: Result4Tuple
          then onlyCakes(getTerm(Change?))
          else T
         fi)
  fi
if BuyCake? := metaXapply(upModule('VENDING-MACHINE, false),
                          T, 'buy-c, none, 0, unbounded, 0)
/\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                          T, 'change, none, 0, unbounded, 0) .
```

Internal strategies: onlyNitems

```
op onlyNitems : Term Qid Nat -> Term .
```

```
ceq onlyNitems(T, Q, N)
```

```
  = if N == 0
```

```
    then T
```

```
    else (if BuyItem? :: Result4Tuple
```

```
      then onlyNitems(getTerm(BuyItem?), Q, sd(N, 1))
```

```
      else (if Change? :: Result4Tuple
```

```
        then onlyNitems(getTerm(Change?), Q, N)
```

```
        else T
```

```
        fi)
```

```
      fi)
```

```
    fi
```

```
  if (Q == 'buy-c or Q == 'buy-a)
```

```
  /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
    T, Q, none, 0, unbounded, 0)
```

```
  /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
    T, 'change, none, 0, unbounded, 0) .
```

Internal strategies: cakesAndApples

```
op buyItem? : Term Qid -> Bool .
```

```
ceq buyItem?(T, Q)
  = if BuyItem? :: Result4Tuple
    then true
    else (if Change? :: Result4Tuple
          then buyItem?(getTerm(Change?), Q)
          else false
         fi)
    fi
if (Q == 'buy-c or Q == 'buy-a)
/\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                          T, Q, none, 0, unbounded, 0)
/\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                          T, 'change, none, 0, unbounded, 0) .
```


Internal strategies: cakesAndApples

```
op cakesAndApples : Term -> Term .
```

```
eq cakesAndApples(T)
= if buyItem?(T, 'buy-c)
  then (if buyItem?(onlyNItems(T, 'buy-c, 1), 'buy-a)
        then cakesAndApples(onlyNItems(onlyNItems(T, 'buy-c, 1),
                                                'buy-a, 1))
        else T
      fi)
  else T
fi .
```

Internal strategies: Examples

```
Maude> reduce in BUYING-STRATS :  
  insertCoin('q.Coin, insertCoin( '$.Coin,  
                                   '___['$.Coin,'$.Coin,'$.Coin, 'q.Coin])) .
```

```
Maude> reduce in BUYING-STRATS :  
  onlyCakes('___['$.Coin,'$.Coin,'$.Coin, 'q.Coin]) .
```

```
Maude> reduce in BUYING-STRATS :  
  onlyNitems('___['$.Coin,'$.Coin,'$.Coin, 'q.Coin], 'buy-a, 3) .
```

```
Maude> reduce in BUYING-STRATS :  
  cakesAndApples('___['$.Coin,'$.Coin,'$.Coin, 'q.Coin]) .
```

Metaprogramming

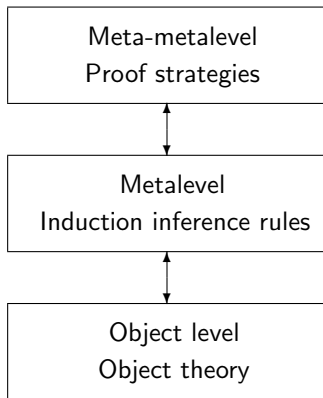
- **Programming at the metalevel**: the metalevel equations and rewrite rules operate on representations of lower-level rewrite theories.
- Reflection makes possible many advanced metaprogramming applications, including
 - user-definable **strategy languages**,
 - language extensions by **new module composition** operations,
 - development of **theorem proving tools**, and
 - reifications of other **languages and logics within rewriting logic**.

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)**.



The ITP tool

Introducing the ITP Tool: a Tutorial

Many thanks

- Our colleagues in the Maude team
 - José Meseguer
 - Steven Eker
 - Francisco Durán
 - Carolyn Talcott
- Our colleagues in the UCMaude group
 - Alberto Verdejo
 - Miguel Palomino
 - Adrián Riesco
- Our colleagues in the world
 - Deepak Kapur
 - Cesare Tinelli