Rewriting Logic and Its Applications in Biology

Part 1: Rewriting Logic and Maude

Temur Kutsia

RISC, Johannes Kepler University of Linz, Austria
`kutsia@risc.uni-linz.ac.at`

May 14, 2009

What Are These Lectures About?

- Describe (yet another) approach to modeling (certain) biological processes.
- The approach is called Pathway Logic.
- The processes it models include signal transduction, metabolism, inter-cellular signalling, neuron systems.
- Pathway logic models are executable in the Maude programming language.
- Maude is based on a simple but powerful logic called Rewriting Logic.

What Are These Lectures About?

- Describe (yet another) approach to modeling (certain) biological processes.
- The approach is called Pathway Logic.
- The processes it models include signal transduction, metabolism, inter-cellular signalling, neuron systems.
- Pathway logic models are executable in the Maude programming language.
- Maude is based on a simple but powerful logic called Rewriting Logic.
- Today: Rewriting Logic and Maude.
- Next week: Pathway Logic.

Contents

How Do We Rewrite?

Example

- Given the rewrite rules:

$$\begin{array}{ll} (1) & 0 + N \rightarrow N \\ (2) & s(M) + N \rightarrow s(M + N) \end{array}$$

- How do we rewrite $s(0) + s(s(0))$?

How Do We Rewrite?

Example

- Given the rewrite rules:

$$\begin{array}{lr} (1) & 0 + N \rightarrow N \\ (2) & s(M) + N \rightarrow s(M + N) \end{array}$$

- How do we rewrite $s(0) + s(s(0))$?

  One rewriting step:

  $s(0) + s(s(0)) \longrightarrow$ (by the rule (2) with $M = 0$, $N = s(s(0))$)
  $s(0 + s(s(0)))$.

How Do We Rewrite?

Example

- Given the rewrite rules:

$$\begin{array}{lr} (1) & 0 + N \rightarrow N \\ (2) & s(M) + N \rightarrow s(M + N) \end{array}$$

- How do we rewrite $s(0) + s(s(0))$?

  Rewrite until impossible (reduce):

$$s(0) + s(s(0)) \longrightarrow \text{(by the rule (2) with } M = 0, N = s(s(0))$$

How Do We Rewrite?

Example

- Given the rewrite rules:

$$\begin{array}{lr} (1) & 0 + N \rightarrow N \\ (2) & s(M) + N \rightarrow s(M + N) \end{array}$$

- How do we rewrite $s(0) + s(s(0))$?

  Rewrite until impossible (reduce):

$$s(0) + s(s(0)) \longrightarrow \text{(by the rule (2) with } M = 0, N = s(s(0))$$
$$s(0 + s(s(0))) \longrightarrow \text{(by the rule (1) with } N = s(s(0))$$

How Do We Rewrite?

Example

- Given the rewrite rules:

$$
\begin{array}{lr}
(1) & 0 + N \rightarrow N \\
(2) & s(M) + N \rightarrow s(M + N)
\end{array}
$$

- How do we rewrite $s(0) + s(s(0))$?

  Rewrite until impossible (reduce):

$$
\begin{array}{l}
s(0) + s(s(0)) \longrightarrow \text{(by the rule (2) with } M = 0, N = s(s(0))\text{)} \\
s(0 + s(s(0))) \longrightarrow \text{(by the rule (1) with } N = s(s(0))\text{)} \\
\quad s(s(s(0))).
\end{array}
$$

Rewriting Logic

What is Rewriting Logic?

- A logic of actions whose models are concurrent systems.
- A logic for executable specification and analysis of software systems.
- A logic to specify other logics or languages.
- An extension of equational logic with local rewrite rules to express
  - concurrent change over time,
  - inference rules.

Rewriting Logic. Some Formal Notions

- Equational specification:
  - Syntax: signature, terms, equations
  - Semantics
- Matching, rewriting
- Rewriting logic (parametrized by an equational specification):
  - Syntax
  - Semantics
  - Inference system

Rewriting Logic. Equational Specification

Many-sorted signature: a pair (S, Σ) where

- S is a set of sorts.
- $\Sigma = \{\Sigma_{\overline{s},s} \mid \overline{s} \in S^*, s \in S\}$ is an $S^* \times S$-sorted family of sets of function symbols.

Rewriting Logic. Equational specification

Example (Many-Sorted Signature)

Let

$$S = \{Nat, Bool\}$$

$$\Sigma = \{\Sigma_{Nat}, \Sigma_{Bool}, \Sigma_{Nat,Nat}, \Sigma_{Nat,Nat,Nat}, \\ \Sigma_{Bool,Bool}, \Sigma_{Bool,Bool,Bool}, \Sigma_{Nat,Nat,Bool}\}$$

where

$$\begin{array}{rcl@{\qquad}rcl} \Sigma_{Nat} & = & \{0\} & \Sigma_{Bool} & = & \{T, F\} \\ \Sigma_{Nat,Nat} & = & \{s\} & \Sigma_{Nat,Nat,Nat} & = & \{+\} \\ \Sigma_{Bool,Bool} & = & \{\neg\} & \Sigma_{Bool,Bool,Bool} & = & \{\vee\} \\ \Sigma_{Nat,Nat,Bool} & = & \{\leq\} & & & \end{array}$$

then (S, Σ) is a many-sorted signature.

Rewriting Logic. Equational specification

S-sorted family of terms

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

over a many-sorted signature (S, Σ) and an S-sorted family $X = \{X_s \mid s \in S\}$ of (pairwise disjoint) sets of variables:

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for each $s \in S$: variables are terms.
2. If $f \in \Sigma_{s_1,\ldots,s_n,s}$, $n \geq 0$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ for each $1 \leq i \leq n$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$: A function symbol applied to terms of the appropriate sorts produces a new term.

Rewriting Logic. Equational specification

Example (Terms)

- (S, Σ) – many-sorted signature from the previous example.
- $X = \{X_{Nat}, X_{Bool}\}$ – family of Variables.
- $x \in X_{Nat}, A \in X_{Bool}$.

Some examples of S-sorted terms:

$$\begin{array}{rclrcl}
x & \in & \mathcal{T}_{\Sigma,Nat}(X) & 0 & \in & \mathcal{T}_{\Sigma,Nat}(X) \\
s(x) & \in & \mathcal{T}_{\Sigma,Nat}(X) & \neg F & \in & \mathcal{T}_{\Sigma,Bool}(X) \\
+(0, s(0)) & \in & \mathcal{T}_{\Sigma,Nat}(X) & \vee(T, A) & \in & \mathcal{T}_{\Sigma,Bool}(X) \\
\leq(s(0), x) & \in & \mathcal{T}_{\Sigma,Bool}(X) & \vee(F, \leq(0, s(0))) & \in & \mathcal{T}_{\Sigma,Bool}(X)
\end{array}$$

Often infix notation is preferred: $F \vee 0 \leq s(0)$ instead of $\vee(F, \leq(0, s(0)))$.

Rewriting Logic. Equational specification

- Notation: $x : s$ means that x is a variable of the sort s.
- Σ-equation:
  $$(x_1 : s_1, \ldots x_n : s_n)\ l = r,$$
  where $l, r \in \mathcal{T}_{\Sigma,s}(\{x_1 : s_1, \ldots x_n : s_n\})$.
- Conditional Σ-equation:
  $$(x_1 : s_1, \ldots x_n : s_n)\ l = r \textit{ if } u_1 = v_1, \ldots, u_m = v_m$$
  where $(x_1 : s_1, \ldots x_n : s_n)\ l = r$, $(x_1 : s_1, \ldots x_n : s_n)\ u_i = v_i$ are Σ-equations.
- Many-sorted specification: (S, Σ, E), where E is a set of conditional Σ-equations.

Rewriting Logic. Equational Specification

- Semantics of many-sorted specification is given by algebras.
- A many-sorted (S, Σ)-algebra consists of a carrier set A_s for each $s \in S$ and a function $F_f^{\bar{s},s} : A_{\bar{s}} \longrightarrow A_s$ for each $f \in \Sigma_{\bar{s},s}$.
- A meaning of a term and satisfaction of a (conditional) equation in an algebra can be defined by induction.
- The semantics of a many-sorted specification (S, Σ, E) is the set of (S, Σ)-algebras that satisfy all (conditional) equations in E.

Rewriting Logic. Equational Specification

- The semantics can be used to answer concrete questions, e.g. whether two terms have the same meaning.
- However, it is more convenient to use syntactic means to answer such questions. It would also provide more opportunities for efficient mechanization.
- Under certain conditions equational deduction can be mechanized by matching and rewriting.
- For rewriting, the equations are oriented from left to right.

Rewriting Logic. Matching and rewriting

Towards matching and rewriting:

- Substitution: A sort-preserving map $\sigma : X \longrightarrow \mathcal{T}_\Sigma(Y)$, where X and Y are S-sorted families of variables for (S, Σ). Substitutions can be uniquely extended to homomorphisms over terms.
- A term t matches a term r with a substitution σ if $\sigma(t) \equiv r$ (syntactically equal).

Matching Rules

- **Trivial (T):**
  $\{t \ll t\} \cup \Gamma; S \Longrightarrow \Gamma; S.$
- **Decomposition (D):**
  $\{f(t_1, \ldots, t_n) \ll f(s_1, \ldots, s_n)\} \cup \Gamma; S \Longrightarrow \{t_1 \ll s_1, \ldots, t_n \ll s_n\} \cup \Gamma; S,$
  if $f(t_1, \ldots, t_n) \not\equiv f(s_1, \ldots, s_n)$.
- **Solve (S):**
  $\{x \ll s\} \cup \Gamma; S \Longrightarrow \Gamma\{x \mapsto s\}; S \cup \{x = s\}.$
- **Symbol Clash (SC):**
  $\{f(t_1, \ldots, t_n) \ll g(s_1, \ldots, s_m)\} \cup \Gamma; S \Longrightarrow \bot.$

Matching Algorithm

In order to match a term *t* to a ground (variable-free) term *s*:

- Create the initial system $\{t \ll s\}, \emptyset$.
- Apply the matching rules as long as it is possible.
- If the process ends with $\emptyset; \{x_1 = r_1, \ldots, x_n = r_n\}$, then success: The substitution $\{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$ matches *t* to *s*.
- If the process ends with $\perp$ then failure: *t* can no match *s*.

Rewriting Logic. Matching and Rewriting

Rewriting with unconditional equations:

- Requirement on oriented equations: All variables in the right hand side also appear in the left hand side.
- Under this assumption, a term t rewrites to a term t' using such an equation $(\cdots)\ l = r$ if
  - there is a subterm q in t such that $q \equiv \sigma(l)$,
  - t' is obtained from t by replacing q with the term $\sigma(r)$

Example (Rewriting)

A term $(0 + s(0)) + y$ rewrites to $s(0) + y$ by the equation $(x : Nat)\ 0 + x = x$.

Rewriting Logic. Matching and Rewriting

Confluence and termination:

- A set of equations E is confluent if the result of rewriting a term is unique: For all t, t_1, t_2 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 if there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \cdots$.
- If E is confluent and terminating, any term t can be reduced to a unique normal form $t \downarrow_E$.
- Efficient mechanization: To check semantic equality of two terms, it is enough to check equality between their respective normal forms.

Example (Confluence and Termination)

$\{(x : Nat)\ 0 + x = 0,\ (x : Nat, y : Nat)\ s(x) + y = s(x + y)\}$ is confluent and terminating (left-to-right rewriting).

Rewriting Logic. Matching and Rewriting

Rewriting with conditional equations:

- Requirement on oriented equations: All variables in the right hand side and in the condition also appear in the left hand side.
- Under this assumption and confluence and termination of E a term t rewrites to a term t' using such an equation $(\cdots)\ l = r\ \mathit{if}\ u_1 = v_1, \ldots, u_n = v_n$ in E if
  - there is a subterm q in t such that $q \equiv \sigma(l)$,
  - $\sigma(u_i) \downarrow_E \equiv \sigma(v_i) \downarrow_E$ for all $1 \leq i \leq n$,
  - t' is obtained from t by replacing q with the term $\sigma(r)$.

Rewriting Logic. Equational Specification

Order-sorted signature:

- Obtained from a many-sorted signature by adding a partial ordering $\leq$ to the set of sorts.
- $s_1 \leq s_2$ is interpreted by the subset inclusion $A_{s_1} \subseteq A_{s_2}$ between the corresponding carrier sets.
- Operations can be overloaded.
- Certain restrictions are introduced to guarantee that each term has the least sort and that equational deduction behaves well.
- Oriented equations should be sort-decreasing.
- Subsorts help to avoid partial functions.

Rewriting Logic. Equational Specification

Example (Ordered Sorts)

The successor function on natural numbers can be used to construct nonzero natural numbers.

Subsorts help to avoid partial functions.

We can define a subsort $NzNat < Nat$, introduce $0 \in \Sigma_{Nat}$, $s \in \Sigma_{Nat,NzNat}$, $div \in \Sigma_{Nat,NzNat,Nat}$ and two equations for it:

$$(x : Nat, y : NzNat)\ x\ div\ y = 0\ if\ y > x$$
$$(x : Nat, y : NzNat)\ x\ div\ y = s((x - y)\ div\ y)\ if\ y \leq x$$

where $>, \leq, -$ are defined elsewhere.

Rewriting Logic

Syntax of Rewriting Logic:

- Signature: an equational specification (Ω, E). RWL is parametrized by the choice its underlying equational logic. For instance, it can be many-sorted or order-sorted equational specification (S, Σ, E), or a more expressive membership equational logic (K, S, Σ, E).
- The signature of RWL makes explicit E in order to emphasize that rewriting will operate on congruence classes modulo E.
- Sentences of RWL are sequents (called rewrites)

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are terms and $[t]_E$, $[t']_E$ are the corresponding congruence classes modulo E.

Rewriting Logic

Syntax of Rewriting Logic:

- A RWL 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 labeled rewrite rules:

  $$r : [t]_E \longrightarrow [t']_E \ if \ [u_1]_E \longrightarrow [v_1']_E \wedge \cdots \wedge [u_1]_E \longrightarrow [v_1']_E,$$

  where t and t' are terms and $[t]_E$, $[t']_E$, etc. are the corresponding congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$ modulo E.

Rewriting Logic

Inference rules of Rewriting Logic (E is omitted from congruence classes for simplicity):

1. **Reflexivity.** For each $[t] \in \mathcal{T}_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

2. **Congruence.** For each $f \in \Sigma_n$,

$$\frac{[t_1] \longrightarrow [t_1'] \cdots [t_n] \longrightarrow [t_n']}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t_1', \ldots, t_n')]}$$

Rewriting Logic

Inference rules of Rewriting Logic:

3. **Replacement.** For each rewrite rule

$$r : [t(\overline{x})] \longrightarrow [t'(\overline{x})] \textit{ if } \\ [u_1(\overline{x})] \longrightarrow [v_1(\overline{x})] \wedge \cdots \wedge [u_k(\overline{x})] \longrightarrow [v_k(\overline{x})]$$

in R with $\overline{x}$ abbreviating $x_1, \ldots x_n$,

$$\frac{\begin{array}{c}[w_1] \longrightarrow [w_1'] \cdots [w_n] \longrightarrow [w_n'] \\ [\sigma(u_1(\overline{x}))] \longrightarrow [\sigma(v_1(\overline{x}))] \cdots [\sigma(u_k(\overline{x}))] \longrightarrow [\sigma(v_k(\overline{x}))]\end{array}}{[\sigma(t(\overline{x}))] \longrightarrow [\sigma'(t'(\overline{x}))]}$$

where $\sigma = \{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\}$ and $\sigma' = \{x_1 \mapsto w_1', \ldots, x_n \mapsto w_n'\}$

Rewriting Logic

Inference rules of Rewriting Logic:

4. Transitivity.

$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

Rewriting Logic

The inference system can be proved sound and complete with respect to RWL semantics. Not discussed here.

Rewriting Logic

Summarizing computational and logical viewpoints for RWL:

State ↔ *Term* ↔ *Proposition*
Transition ↔ *Rewriting* ↔ *Deduction*
Distributed structure ↔ *Algorithmic structure* ↔ *Propositional structure*

Maude

- Maude is a language and environment based on rewriting logic.
- See: http://maude.cs.uiuc.edu/
- Features:
  - Executability — position /rule/object fair rewriting
  - High performance engine — {ACI} matching
  - Modularity and parameterization
  - Builtins — booleans, number hierarchy, strings
  - Reflection – using descent and ascent functions
  - Search and model-checking

Modules in Maude

- Module - the key concept of Maude.
- Modules define a collection of operations and how they interact.
- Three types of modules:
  - functional module (`fmod`).
  - system module (`mod`).
  - object-oriented module (`omod`).
- A module can be imported from another.

Sorts

- A sort is declared within the module.
- `sort integer .`
-
```
sorts Real Irrational Rational Integer
      Fraction Positive Negative .
subsorts Irrational Rational < Real .
subsort Fraction < Rational .
subsorts Positive Negative < Integer < Rational .
```

Variables

- Variables are declared within the module.
-
  ```
  var x : number .
  vars c1 c2 c3 : color .
  ```

Operations

Addition operator adding two natural numbers (sort `Nat`):

- prefix declaration: `op + :  Nat Nat -> Nat .`
- mixfix declaration: `op _+_ :  Nat Nat -> Nat .`

Two operations with the same sort arguments and sort results can be declared by using the key word `ops`:

- Prefix decl.: `ops + * :  Nat Nat -> Nat .`
- Mixfix decl.: `ops _+_ _*_ :  Nat Nat -> Nat .`

Operator overloading is allowed.

Operators

The Peano notation of natural numbers:

```
fmod PEANO-NAT is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor] .
endfm
```

- `0`, `s` are constants. Argument sort is not given.
- `ctor` stands for constructor.

Operators

Lists:

```
fmod BASIC-LIST is
    sorts List Elt .
    subsort Elt < List .
    op nil : -> List [ctor] .
    op __ : List List -> List [ctor assoc id: nil] .
    vars E1 E2 : Elt . vars L1 L2 : List .
endfm
```

- Concatenation operation __ is in mixfix notation, and would be called with variables `L1` and `L2` as “`L1 L2`”.
- `[ctor assoc id: nil]`: Concatenation is a constructor, is associative, and has `nil` as the identity element.
- Other useful attributes: `comm` (commutativity), `idem` (idempotency).

Built-in Modules

- `NAT, INT, FLOAT, STRING, ...`
- `QID`: Quoted identifiers.
- Create a sort `Name` without defining constants of this sort:

  ```
  fmod NAME is
      protecting QID .
      sort Name .
      subsort Qid < Name .
  endfm
  ```

  Any quoted identifier (e.g. `'john`) becomes a constant of sort `Name`, without declaring them explicitly.

Equations

- Idea of equations: to provide the Maude interpreter with certain rules to simplify an expression.

```
fmod PEANO-NAT-EXTRA is
    sort Nat .
    op 0 : -> Nat [ctor] .
    op s : Nat -> Nat [ctor iter] .
    op _+_ : Nat Nat -> Nat .
    vars M N : Nat .
    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

Equations

- Variables can be defined on the fly, if they are used only once.

  ```
  op n_t_pd_l_ :
    Name Title PublDate Loc -> CatCard [ctor] .
  op author :  CatCard -> Name .
  ```

  If only one equation uses these sorts, one does not need to declare variables separately:

  ```
  eq author
    (
     n N:Name t T:Title pd P:PublDate l L:Loc
    )
  = N:Name .
  ```

Recursion

- Most important strategy for writing equations that simplify well.
- Nearly every set of equations is defined with some level of recursion in mind.
- If you don't like recursion, Maude is definitely not the programming language for you.
-
  ```
      eq 0 + N = N .
      eq s(M) + N = s(M + N) .
  ```
- The second equation employs recursion by calling `_+_` again on the right hand side.
- The first equation ends the recursion.

Recursion

- Basic module for lists:

```
fmod LIST is
  sorts List Elt .
  subsort Elt < List .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil]
.
endfm
```

- Add an operator `size` that computes the size of a list:

```
fmod LIST-SIZE is
  protecting LIST .
  protecting PEANO-NAT .
  op size : List -> Nat .
  var E : Elt . var L : List .
  eq size(nil) = 0 .
  eq size(E L) = s(size(L)) .
endfm
```

Conditional Equations

- Conditional equations: equations that depend on a Boolean statement.
- Written with the key word `ceq` and, after the equation, a condition starting with the key word `if`.
- Maude comes with a sort `Bool` along with constants `true` and `false`, and `==`, `=/=`, `and`, `or`, and `not` operations (all mixfix).
- A conditional equation will execute a reduction only if its condition reduces to `true`.
- `ceq different?(N, M) = true if N =/= M .`

`if_then_else_fi`

- `if_then_else_fi` is an operator provided by Maude.
- It is not necessary to declare equations with this operator as conditional, because they aren't using the key word `if`.
- `eq max(M, N) = if N > M then N else M fi .`

Pattern Matching in Conditions

- One of the most powerful features of Maude: to use a pattern match as a condition, using the key symbol `:=`.
- This symbol compares a pattern on the left-hand side with the right-hand side, and if they match, returns `true`.

Pattern Matching in Conditions

- Example: An irritable professor takes questions only after class and reacts angrily on any other sort of interruption:

-
  ```
  fmod IRRITABLE-PROFESSOR is
    protecting STRING .
    sorts Question Exclamation Interruption .
    subsorts Question Exclamation < Interruption .
    op _? : String -> Question [ctor] .
    op _! : String -> Exclamation [ctor] .
    op reply : Interruption -> String .
    var I : Interruption .
    ceq reply(I) = "Questions after class, please"
         if (S:String) ? := I .
    eq reply(I) = "Shut up!" [owise] .
  endfm
  ```

The Maude Environment

- Write the modules for the algebras to be used in the environment.
- Store them in a file directory that the Maude environment can access.
- Load the needed module by typing `load MODULE-NAME` into the prompt.
- Type in `reduce` (or `red`), followed by the expression to be reduced. For example:

  ```
  Maude> reduce s(0) + s(s(0)) .
  result Nat:  s(s(s(0)))
  ```

The Maude Environment

- To change the current module, type in `select MODULE-NAME` into the prompt.
- Alternatively, one may specify the required module name with `in MODULE-NAME`:

  ```
  Maude> red in PEANO-NAT-MULT : s(s(0)) *
  s(s(s(0))) .
  result Nat:  s(s(s(s(s(s(0)))))) .
  ```
- Activating tracing:

  ```
  Maude> set trace on .
  ```

Rewrite Rules

- The real power of Maude is about transitions that occur within and between structures.
- These transitions are mapped out in rewrite laws.
- Rewriting logic consists of two key ideas: states and transitions.
- States are static situations.
- Transitions are the transformations that map one state to another.
- Example:

```
mod CLIMATE is
  sort weathercondition .
  op sunnyday : -> weathercondition .
  op rainyday : -> weathercondition .
  rl [raincloud] : sunnyday => rainyday .
endm
```

Rewrite Rules

Example

- A certain hobo can make one cigarette out of four cigarette butts (what’s left after smoking a cigarette).
- If the hobo starts off with sixteen cigarettes, how many cigarettes can he smoke in total?

Rewrite Rules

Example

- A certain hobo can make one cigarette out of four cigarette butts (what's left after smoking a cigarette).
- If the hobo starts off with sixteen cigarettes, how many cigarettes can he smoke in total?
- 21.
- Once he smokes 16 cigarettes, he can make the 16 butts into 4 more cigarettes.
- Once he smokes those, he can make the 4 butts into 1 more cigarette.

Rewrite Rules

Example (Cont.)

Simple Implementation (Does not compute, just checks):

```
mod CIGARETTES is
  sort State .
  op c : -> State [ctor] . *** cigarette
  op b : -> State [ctor] . *** butt
  op __ : State State -> State [ctor assoc comm]
.
  rl [smoke] : c => b .
  rl [makenew] : b b b b => c .
endm
```

- To rewrite, use Maudes `rewrite` (or `rew`):
- `rew [100] c c c c c c c c c c c c c c c c`

Rewrite Rules

Example (Bigger Example: Toy Crane)

- Blocks world, the basic idea: Create an algorithm for a bunch of blocks stacked on each other or the table, and a robot arm that can carry them.
- This example: Toy crane machines with stuffed animals in a big glass box and a controllable arm that moves and tries to grab them.

Rewrite Rules

Example (Bigger Example: Toy Crane)

- For stuffed animals, there are three state constructors needed:
  1. the animal is on the floor of the machine, not on top of any other animal.
  2. The stuffed animal is on top of another stuffed animal.
  3. The stuffed animal is clear, that is, there are no other stuffed animals on top of it.
- For the robot arm/claw/crane, there are two:
  1. the claw is holding a stuffed animal.
  2. it is empty.

Rewrite Rules

Example (Bigger Example: Toy Crane)

- Transitions:
  - pick up from the floor.
  - put down on the floor.
  - unstack (pick up from the top of another animal).
  - stack (put down on another animal).

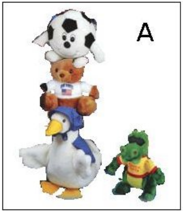
Rewrite Rules

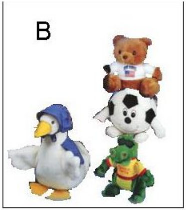
Example (Bigger Example: Toy Crane)

```
mod ARCADE-CRANE is
  protecting QID .
  sorts ToyID State .
  subsort Qid < ToyID .
  op floor : ToyID -> State [ctor] .
  op on : ToyID ToyID -> State [ctor] .
  op clear : ToyID -> State [ctor] .
  op hold : ToyID -> State [ctor] .
  op empty : -> State [ctor] .
  op 1 : -> State [ctor] .
*** this is the identity State; it’s just good to have one.
  op _&_ : State State -> State [ctor assoc comm id: 1] .
  vars X Y : ToyID .
  rl [pickup] : empty & clear(X) & floor(X) => hold(X) .
  rl [putdown] : hold(X) => empty & clear(X) & floor(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
```

Rewrite Rules

Example (Bigger Example: Toy Crane)

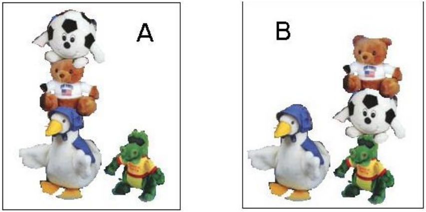




- State A: `empty & floor(’mothergoose) & on(’teddybear,’mothergoose) & on(’soccerball,’teddybear) & clear(’soccerball) & floor(’dragondude) & clear(’dragondude).`
- State B: `empty & floor(’mothergoose) & clear(’mothergoose) & floor(’dragondude) & on(’soccerball, ’dragondude) & on(’teddybear, ’soccerball) & clear(’teddybear).`

Rewrite Rules

Example (Bigger Example: Toy Crane)



- The set of transitions between these two states:
    - `unstack` called on `'soccerball`
    - `stack` called on ’`soccerball` and `'dragondude`
    - `unstack` called on `'teddybear`
    - `stack` called on `'teddybear` and `'soccerball`.

Rewrite Rules

Example (Bigger Example: Toy Crane)

Exercise:

- Enter `ARCADE-CRANE` module into the Maude environment.
- Call `rew [2]` and `rew [4]`. Explain the results.
- Call `frew [30]` (fair rewrite).

Rewrite Rules

Example (Bigger Example: Toy Crane)

Exercise:

- Call
  ```
  search in ARCADE-CRANE : empty &
  floor(’mothergoose) & on(’teddybear,
  ’mothergoose) & on(’soccerball, ’teddybear) &
  clear(’soccerball) & floor(’dragondude) &
  clear(’dragondude) =>+ empty &
  floor(’teddybear) & on
  (’mothergoose,’teddybear) &
  on(’soccerball,’mothergoose) &
  clear(’soccerball) & floor(’dragondude) &
  clear(’dragondude) .
  ```
- Explain the input and the output.

Rewrite Rules

Example (Bigger Example: Toy Crane)

Exercise:

- Call

  ```
  search in ARCADE-CRANE : empty &
  floor('mothergoose) &
  on('teddybear,'mothergoose) &
  on('soccerball,'teddybear) & clear('soccerball)
  & floor('dragondude) & clear('dragondude) =>+
  empty & floor('teddybear) & floor('mothergoose)
  & M:State .
  ```

- Explain the input and the output.

Rewrite Rules

Exercise:

- Modify `mod CIGARETTES` so that it solves the problem, computing the number of cigarettes.

References

Materials used to prepare these lectures:

- Papers on Maude and Rewriting Logic.
  http://maude.cs.uiuc.edu/papers/.
- Pathway Logic web page.
  http://pl.csl.sri.com/.
- Maude Primer
  http://maude.cs.uiuc.edu/primer/.
- M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada.
  A Maude Tutorial.
  SRI International, 2000.

References

- S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott.
  Pathway Logic: Executable models of biological networks.
  ENTCS, 71, 2002.
- J. Meseguer.
  Bio-Pathway Logic. Slides.
- J. Meseguer.
  Conditional Rewriting Logic as a unified model of concurrency.
  TCS, 96(1):73–155, 1992.
- C. Talcott.
  Pathway Logic tutorial. Parts 1 and 2. Slides.