Program Verification: Lecture 14

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

The ITP Inference Rules

Notice that in the ITP we reason backwards, replacing the main goal G we want to prove by subgoals, $G_1, \ldots, G_n$, such that if we prove each of the subgoals, then we have proved the main goal.

For such an inference to be sound, the implication

$$G_1 \wedge \ldots \wedge G_n \Rightarrow G$$

should always be satisfied, that is, should be semantically valid in the initial algebra $\mathbb{T}_{\Sigma,E}$ on which we are doing the inductive reasoning.

The ITP Inference Rules (II)

Such semantically valid inferences are expressed as inference rules

$$\frac{G_1 \quad \ldots \quad G_n}{G}$$

However, since we are reasoning backwards, from the root of the proof tree to the leaves, the ITP uses such rules in the opposite direction, as rules

$$\frac{G}{G_1 \quad \ldots \quad G_n}$$

We will illustrate through an example such backward reasoning for the ITP induction inference rule, and will at the same time justify its soundness.

Induction on Other Data Structures: Tree Induction

We have already seen examples of how the ITP's `ind` rule applies to natural number induction and to list induction.

Before discussing the most general form of the `ind` rule for any signature of constructors Ω and its justification, we give an example illustrating binary tree induction, in which the data in leaves are seen as depth-zero trees.

The intuitive idea is that to prove an inductive property P about such trees we must show: (1) that P holds for the data elements (**base case**); and (2) that if P holds for the left and right subtrees, then it must hold for their binary join (**induction step**).

Induction on Other Data Structures: Tree Induction (II)

Consider the following module defining binary trees whose nodes are quoted identifiers (constants in the predefined module QID), and a reverse function on binary trees.

```
fmod TREE is
protecting QID .
sort Tree .
subsort Qid < Tree .
op _#_ : Tree Tree -> Tree [ctor] .
op rev : Tree -> Tree .
var I : Qid .
vars T T' : Tree .
eq rev(I) = I .
eq rev(T # T') = rev(T') # rev(T) .
endfm
```

Induction on Other Data Structures: Tree Induction (III)

We can apply binary tree induction to prove that for all trees `T` the equation `rev(rev(T)) = T` holds. We can do so by entering the `TREE` module in the ITP and the goal:

```
Maude> (goal rev : TREE |- A{T:Tree}((rev(rev(T:Tree)) = (T:Tree)) .)

=================================
label-sel: rev@0
=================================
A{T:Tree} rev(rev(T:Tree)) = T:Tree
+++++++++++++++++++++++++++++++++
```

Induction on Other Data Structures: Tree Induction (IV)

We can then try to prove this goal by induction on `T:Tree`.

```
Maude> (ind on T:Tree .)



=================================
label-sel: rev@1.0
=================================
A{V0#0:Qid} rev(rev(V0#0:Qid)) = V0#0:Qid ==>
rev(rev(V0#0:Qid)) = V0#0:Qid

=================================
label: rev@2.0
=================================
A{V0#0:Tree ; V0#1:Tree} rev(rev(V0#1:Tree)) = V0#1:Tree &
rev(rev(V0#0:Tree)) = V0#0:Tree ==>
rev(rev(V0#0:Tree # V0#1:Tree)) = V0#0:Tree # V0#1:Tree
+++++++++++++++++++++++++++++++++
```

Induction on Other Data Structures: Tree Induction (V)

Note that goal `rev@2.0` is the "induction step" in tree induction, whereas the "base case" is goal `rev@1.0`. Both subgoals can then be proved using the `auto` tactic.

```
Maude> (auto .)

=================================
label-sel: rev@2.0
=================================
A{V0#0:Tree ; V0#1:Tree} rev(rev(V0#1:Tree)) = V0#1:Tree &
rev(rev(V0#0:Tree)) = V0#0:Tree ==>
rev(rev(V0#0:Tree # V0#1:Tree)) = V0#0:Tree # V0#1:Tree
+++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d
```

Structural Induction

We have already observed how the ITP supports inductive proofs in three cases: natural number induction, list induction, and tree induction. But what is the general form of induction supported by the ITP for a specification having a subsignature Ω of constructors? This general form is called structural induction. It reduces proving an inductive property of the form $(\forall x : s)\ P(x)$, to proving:

- **Base Case**. For any constant $a : nil \longrightarrow s'$ in Ω with $s' \leq s$, the subgoal $P(x \mapsto a)$

  Notation: Given a variable x, the substitution $\{x \mapsto t\}$ mapping x to a term t is abbreviated to $(x \mapsto t)$, and its homomorphic extension is denoted $\_(x \mapsto t)$.

Structural Induction (II)

- **Induction Step**. For each constructor $f : s_1 \ldots s_{n_f} \longrightarrow s'$ in Ω with $s' \leq s$, where the sorts $s_{i_1}, \ldots, s_{i_{k_f}}$ are those among the $s_1 \ldots s_{n_f}$ such that $s_{i_j} \leq s$, $1 \leq j \leq k_f$, the subgoal,

$$(\forall x_1 : s_1, \ldots, x_{n_f} : s_{n_f}) \bigwedge_{1 \leq j \leq k_f} P(x \mapsto x_{i_j}) \rightarrow P(x \mapsto f(x_1, \ldots, x_{n_f})).$$

Note: It may happen that none of the sorts among the $s_1 \ldots s_{n_f}$ is s or a subsort of s. In that case, the subgoal has the form $(\forall x_1 : s_1, \ldots, x_{n_f} : s_{n_f})\, P(x \mapsto f(x_1, \ldots, x_{n_f}))$.

Structural Induction (III)

Structural Induction is an inference rule of the form,

$$\frac{\bigwedge_i P(x \mapsto a_i) \wedge \bigwedge_l (\forall x_1, \ldots, x_{n_{f_l}}) \; \bigwedge_{1 \leq j \leq k_{f_l}} P(x \mapsto x_{i_j}) \Rightarrow P(x \mapsto f_l(x_1, \ldots, x_{n_{f_l}}))}{(\forall x : s) \; P(x)}$$

where the a_i and the f_j include all the constructor constants and operators meeting the properties specified above.

In the ITP this rule is used backwards as the `ind` rule,

$$\frac{(\forall x : s) \; P(x)}{\bigwedge_i P(x \mapsto a_i) \wedge \bigwedge_l (\forall x_1, \ldots, x_{n_{f_l}}) \; \bigwedge_{1 \leq j \leq k_{f_l}} P(x \mapsto x_{i_j}) \Rightarrow P(x \mapsto f_l(x_1, \ldots, x_{n_{f_l}}))}$$

Justification of the `ind` Rule

Why is `ind` a sound inference rule? First consider:

Lemma: For (Σ, E) confluent, sort-decreasing, terminating and sufficiently complete for constructors Ω, given any Σ-equation $t = t'$ with $X = vars(t = t')$ we have:

$$\mathbb{T}_{\Sigma/E} \models t = t' \quad \Leftrightarrow \quad \forall \theta \in [X \to T_\Omega]\ \mathbb{T}_{\Sigma/E} \models t\theta = t'\theta.$$

Proof: Since $\mathbb{T}_{\Sigma/E} \cong \mathbb{C}_{\Sigma/E}$ it is enough to prove that

$$\mathbb{C}_{\Sigma/E} \models t = t' \quad \Leftrightarrow \quad \forall \theta \in [X \to T_\Omega]\ \mathbb{C}_{\Sigma/E} \models t\theta = t'\theta.$$

But, since $C_{\Sigma/E} \subseteq T_\Omega$, any $a : X \longrightarrow C_{\Sigma/E}$ is a substitution $\theta : X \longrightarrow T_\Omega$, exactly one of the form $\theta = \theta!_E$. Furthermore, for each $\theta \in [X \to T_\Omega]$ we have the equivalence,

$$\mathbb{C}_{\Sigma/E} \models t\theta = t'\theta \quad \Leftrightarrow \quad (t\theta)!_E = (t(\theta!_E))!_E = (t'(\theta!_E))!_E = (t'\theta)!_E.$$

Justification of the `ind` Rule (II)

But since any $\theta : X \longrightarrow C_{\Sigma/E}$ satisfies $\theta = \theta!_E$, $\forall\, \theta \in [X \to T_\Omega]\ (t(\theta!_E))!_E = (t'(\theta!_E))!_E$ exactly means $\mathbb{C}_{\Sigma/E} \models t = t'$. q.e.d.

Notice that the above Lemma easily generalizes to the modulo B case, that is, to theories $(\Sigma, E \cup B)$ with E ground confluent, sort-decreasing, terminating and sufficiently complete for Ω modulo B and Σ preregular modulo B. Our justification of the `ind` rule in what follows works just the same for the modulo B case.

Justification of the `ind` Rule (III)

Notice that the argument of the above lemma does not depend on our formula being actually an equation: by reasoning inductively on the structure of formulas we can show that the lemma applies to any universally-quantified first-order formula of the form $(\forall x : s)\ P(x)$ (P itself can have other quantifiers).

Therefore, we have reduced the problem of proving an inductive property, $(\forall x : s)\ P(x)$, to that of proving that for all $t \in T_{\Omega,s}$ the instantiated property $P(x \mapsto t)$ holds.

Here is where structural induction steps in as a method, namely, by analyzing more closely what it means to prove something for all $t \in T_{\Omega,s}$.

Justification of the `ind` Rule (IV)

Theorem. (*Soundness of Structural Induction*). For (Σ, E) ground confluent, sort-decreasing, and terminating with subsignature of constructors Ω, if we have

$$\mathbb{T}_{\Sigma/E} \models \bigwedge_i P(x \mapsto a_i) \wedge \bigwedge_l (\forall x_1, \ldots, x_{n_{f_l}}) \bigwedge_{1 \leq j \leq k_{f_l}} P(x \mapsto x_{i_j}) \Rightarrow P(x \mapsto f_l(x_1, \ldots, x_{n_{f_l}}))$$

then we also have

$$\mathbb{T}_{\Sigma/E} \models (\forall x : s)\ P(x).$$

Proof. Suppose not. I.e., the hypothesis holds and there is a ground constructor term $t \in T_{\Omega,s}$ s.t. $\mathbb{T}_{\Sigma/E} \not\models P(x \mapsto t)$. Choose such $t \in T_{\Omega,s}$ of smallest depth possible. That is any other $t' \in T_{\Omega,s}$ such that $\mathbb{T}_{\Sigma/E} \not\models P(x \mapsto t')$ must have tree depth greater or equal to that of t.

Justification of the `ind` Rule (V)

Suc a term t cannot be a constant a_i of sort less or equal to s, since we have $\mathbb{T}_{\Sigma/E} \models \bigwedge_i P(x \mapsto a_i)$. Therefore, t must be of the form $t = f_q(t_1, \ldots, t_{n_{f_q}})$. But by the minimal depth assumption on t, we must have $\mathbb{T}_{\Sigma/E} \models P(x \mapsto t_{i_j})$, $1 \leq j \leq k_{f_q}$. Which by the theorem's hypothesis implies $\mathbb{T}_{\Sigma/E} \models P(x \mapsto f_q(t_1, \ldots, t_{n_{f_q}}))$. That is, $\mathbb{T}_{\Sigma/E} \models P(x \mapsto t)$, contradicting the assumption $\mathbb{T}_{\Sigma/E} \not\models P(x \mapsto t)$. q.e.d.

Verification of Concurrent Programs

We will begin considering the topic of verification of concurrent programs. We will consider first the case of declarative concurrent programs. Later in the course we will also consider verification of imperative (sequential or concurrent) programs.

So the first question is, what is a suitable computational logic to write concurrent programs in a declarative style? This is of course an open-ended question, in that a variety of answers are possible at present, and new answers may be proposed in the future.

Verification of Concurrent Programs (II)

In this course, we will use rewriting logic as a specific computational logic that is indeed well suited for concurrent programming.

This is in full harmony with our use of equational logic for what, rather than sequential, we could better call deterministic declarative programming. In fact, rewriting logic generalizes equational logic in a natural way.

Rewrite Theories: Preliminary Definition

We give a first, already quite general, definition of rewrite theories. We will further generalize this notion later.

A rewrite theory $\mathcal{R}$ is a triple $\mathcal{R} = (\Sigma, E, R)$, with:

- (Σ, E) a (kind-complete) order-sorted equational theory, and
- R a set of labeled rewrite rules of the form $l : t \longrightarrow t' \;\; \Leftarrow cond$, with l a label, $t, t' \in T_{\Sigma}(X)_k$ for some kind k, and $cond$ a condition (involving the same variables X) as explained below.

Conditional Rewrite Rules

The most general form of a conditional rewrite rule is:

$$l : t \longrightarrow t' \ \Leftarrow \ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j w_j \longrightarrow w_j'),$$

that is, in general, the condition is a conjunction of equations and rewrites, where the variables in all the Σ-terms $t, t', u_i, u_i', w_j, w_j'$ are contained in a common set X. There is no requirement that $vars(t) = X$, and no assumptions of confluence or termination. The rule is called unconditional if the condition is empty.

Maude System Modules

In Maude, rewrite theories are specified in system modules.

The same way that a functional module has essentially the form, `fmod` (Σ, E) `endfm`, with (Σ, E) an order-sorted equational logic theory, a system module has the form, `mod` (Σ, E, R) `endm`, with (Σ, E, R) a rewrite theory.

We will illustrate the syntax details in examples. In particular, a conditional rewrite rule of the form, $l : t \longrightarrow t' \quad \Leftarrow cond$ is specified in Maude with syntax,

$$\texttt{crl [}l\texttt{] : } t \texttt{ => } t' \texttt{ if } cond \texttt{ .}$$

and an unconditional rule $l : t \longrightarrow t'$ with syntax,

$$\texttt{rl [}l\texttt{] : } t \texttt{ => } t' \texttt{ .}$$

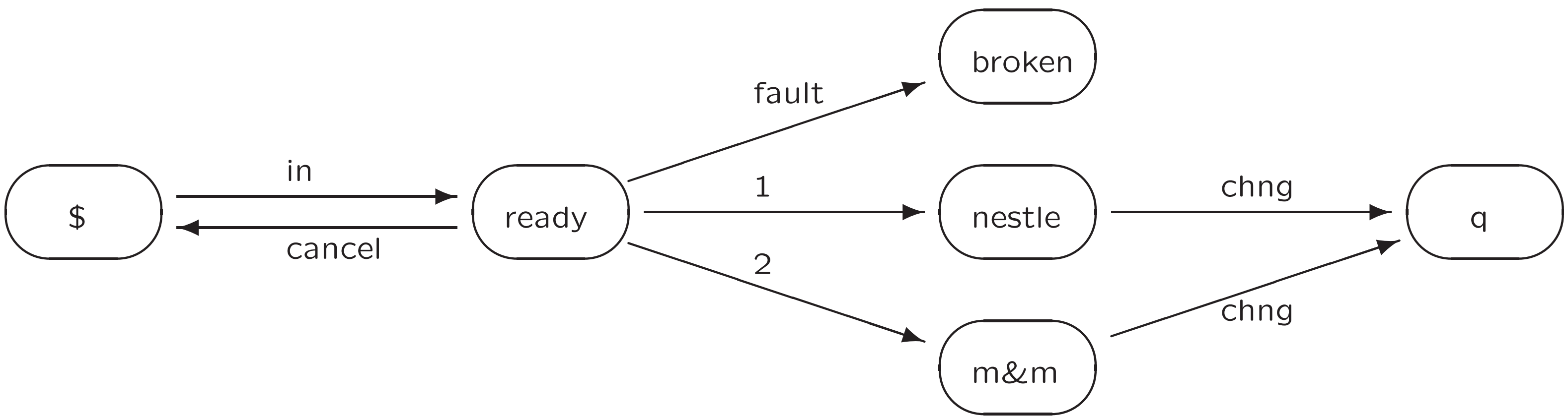
Some Rewriting Logic Examples

To motivate rewriting logic as a formalism to mathematically model and program concurrent systems, we will show how it can be used to naturally specify three important classes of systems, namely:

- automata, also called labeled transition systems,
- Petri nets, one of the simplest concurrency models, and
- object-oriented concurrent systems.

Concurrency vs. Nondeterminism: Automata

We can motivate concurrency by its absence. The point is that we can have systems that are nondeterministic, but are not concurrent. Consider the following faulty automaton to buy candy:



Concurrency vs. Nondeterminism: Automata (II)

Although in the above automaton each labeled transition from each state leads to a single next state, the automaton is nondeterministic in the sense that the automaton's computations are not confluent, and therefore completely different outcomes are possible.

For example, from the `ready` state the transitions `fault` and `1` lead to completely different states that can never be reconciled in a common subsequent state.

Concurrency vs. Nondeterminism: Automata (III)

So, the automaton is in this sense nondeterminisitc, yet it is strictly sequential, in the sense that, although at each state the automaton may be able to take several transitions, it can only take one transition at a time.

Since the intuitive notion of concurrency is that several transitions can happen simultaneously, we can conclude by saying the our automaton, although it exhibits a form of nondeterminism, has no concurrency whatsoever.

Automata as Rewrite Theories

We can specify such an automaton as a system module,

```
mod CANDY-AUTOMATON is
  sort State .
  ops $ ready broken nestle m&m q : -> State .
  rl [in] : $ => ready .
  rl [cancel] : ready => $ .
  rl [1] : ready => nestle .
  rl [2] : ready => m&m .
  rl [fault] : ready => broken .
  rl [chng] : nestle => q .
  rl [chng] : m&m => q .
endm
```

Rewite Rules as Transitions

Note that rewrite rules do not have an equational interpretation. They are not understood as equations, but as transitions, that in general cannot be reversed.

This is why, in a rewite theory (Σ, E, R) the equations in E are totally different from the rules R, since equations and rules have a totally different semantics.

However, operationally Maude will assume that the equations in E are confluent, terminating, and sort decreasing modulo axioms B, and will compute with such equations and also with the rules in R by rewriting, yet distinguishing equation simplification (the `reduce` command) from rewriting with rules (the `rewrite` command).

The `rewrite` Command

Maude can execute rewrite theories with the `rewrite` command (can be abbreviated to `rew`). For example,

```
Maude> rew $ .
rewrite in CANDY-AUTOMATON : $ .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result State: q
```

The `rewrite` command applies the rules in a fair way (all rules are given a chance) hopefully until termination, and, if it terminates, gives one result.

The `rewrite` Command (II)

In this example, fairness saves us from nontermination, but in general we can esily have nonterminating computations.

For this reason the `rewrite` command can be given a numeric argument stating the maximum number of rewrite steps. Furthermore, using Maude's the `trace` command we can observe such steps. For example,

The `rewrite` Command (III)

```
Maude> set trace on .
Maude> rew [3] $ .
rewrite [3] in CANDY-AUTOMATON : $ .
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
*********** rule
rl [cancel]: ready => $ .
empty substitution
ready ---> $
*********** rule
rl [in]: $ => ready .
empty substitution
$ ---> ready
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result State: ready
```

The `search` Command

Of course, since we are in a nondeterministic situation, the `rewrite` command gives us one possible behavior among many.

To systematically explore all behaviors from an initial state we can use the `search` command, which takes two terms: a ground term which is our initial state, and a term, possibly with variables, which describes our desired target state.

Maude then does a breadth first search to try to reach the desired target state. For example, to find the terminating states from the `$` state we can give the command (where the "`!`" in `=>!` specifies that the target state must be a terminating state),

The `search` Command (II)

```
Maude> search $ =>! X:State .
search in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken

Solution 2 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

We can then inspect the search graph by giving the command,

The `search` Command (III)

```
Maude> show search graph .
state 0, State: $
arc 0 ===> state 1 (rl [in]: $ => ready .)

state 1, State: ready
arc 0 ===> state 0 (rl [cancel]: ready => $ .)
arc 1 ===> state 2 (rl [1]: ready => nestle .)
arc 2 ===> state 3 (rl [2]: ready => m&m .)
arc 3 ===> state 4 (rl [fault]: ready => broken .)

state 2, State: nestle
arc 0 ===> state 5 (rl [chng]: nestle => q .)

state 3, State: m&m
arc 0 ===> state 5 (rl [chng]: m&m => q .)

state 4, State: broken
state 5, State: q
```

The `search` Command (IV)

We can then ask for the shortest path to any state in the state graph (for example, state 5) by giving the command,

```
Maude> show path 5 .
state 0, State: $
===[ rl [in]: $ => ready . ]===>
state 1, State: ready
===[ rl [1]: ready => nestle . ]===>
state 2, State: nestle
===[ rl [chng]: nestle => q . ]===>
state 5, State: q
```

The `search` Command (V)

Similarly, we can search for target terms reachable by one or more rewrite steps, or zero or more steps by typing (respectively):

- `search` t `=>+` t' .
- `search` t `=>*` t' .

The `search` Command (VI)

Furthermore, we can restrict any of those searches by giving an equational condition on the target term. For example, all terminating states reachable from `$` other than `broken` can be found by the command,

```
Maude> search $ =>! X:State such that X:State =/= broken .
search in CANDY-AUTOMATON : $ =>! X:State
such that X:State =/= broken = true .

Solution 1 (state 5)
states: 6 in 0ms cpu (0ms real)
X:State --> q
```

The `search` Command (VII)

Of course, in general there can be an infinite number of solutions to a given search. Therefore, a search can be further restricted by giving as an extra parameter in brackets the number of solutions (i.e., target terms that are instances of the pattern and satisfy the condition) we want:

```
search [1] in CANDY-AUTOMATON : $ =>! X:State .

Solution 1 (state 4)
states: 6 in 0ms cpu (0ms real)
X:State --> broken
```

The `search` Command (VIII)

In our CANDY-AUTOMATON example the number of states is finite, but for a general rewrite theory the number of states reachable from an initial state can be infinite. So, even if we search for a single solution, the search process may not terminate, because no such solution exists. To make search terminating, at least for unconditional rewrite rules, we can add a second parameter, namely, a bound on the length of the paths searched from the initial state.

```
search [1, 1] in CANDY-AUTOMATON : $ =>! X:State .

No solution.
states: 2  rewrites: 2 in 0ms cpu (36ms real) (~ rewrites/second)
```

Labelled Transition Systems

Our `CANDY-AUTOMATON` example is just a special instance of a general concept, namely, that of automaton, also called a labeled transition system (LTS) by which we mean a triple: $\mathcal{A} = (A, L, T)$ with:

- A is a set, called the set of states,
- L is a set called the set of labels, and
- $T \subseteq A \times L \times A$ is called the set of labeled transitions.

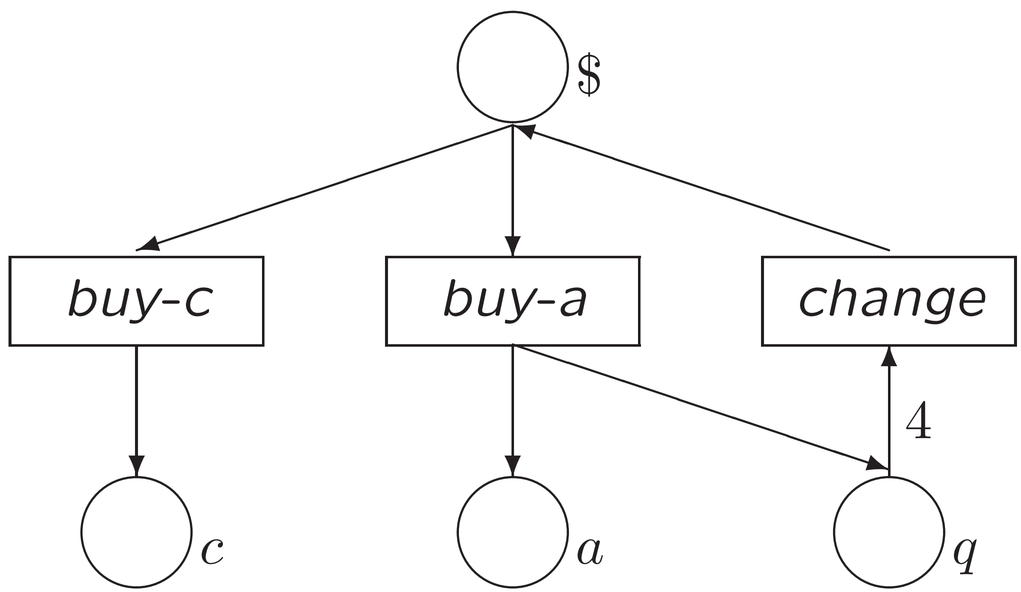
LTS's as Rewrite Theories

Note that we have associated to our candy automaton a rewrite theory (system module) `CANDY-AUTOMATON`.

This is of course just an instance of a general transformation, that assign to a LTS A a rewrite theory $R(A)$ with a single sort A, constants $x \in A$, and for each $(x, l, y) \in T$ a rewrite rule $l : x \longrightarrow y$.

Petri Nets

So far so good, but we have not yet seen any concurrency. The simplest concurrent system examples are probably the concurrent automata called Petri nets. Consider for example the picture,



Petri Nets (II)

The previous picture represents a concurrent machine to buy cakes and apples; a cake costs a dollar and an apple three quarters.

Due to an unfortunate design, the machine only accepts dollars, and it returns a quarter when the user buys an apple; to alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is concurrent, because we can push several buttons at once, provided enough resources exist in the corresponding slots, which are called places

Petri Nets (III)

For example, if we have one dollar in the $\$$ place, and four quarters in the q place, we can simultaneously push the *buy-a* and *change* buttons, and the machine returns, also simultaneously, one dollar in $\$$, one apple in a, and one quarter in q.

That is, we can achieve the concurrent computation,

$$\textit{buy-a}\ \ \textit{change} : \$\, q\, q\, q\, q \longrightarrow a\, q\, \$.$$

Petri Nets (IV)

This has a straightforward expression as a rewrite theory (system module) as follows:

```
mod PETRI-MACHINE is
  sort Marking .
  ops null $ c a q : -> Marking .
  op _ _ : Marking Marking -> Marking [assoc comm id: null] .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [chng] : q q q q => $ .
endm
```

Petri Nets (V)

That is, we view the distributed state of the system as a multiset of places, called a marking, with identity for multiset union the empty multiset `null`.

We then view a transition as a rewite rule from one (pre-)marking to another (post-)marking.

Petri Nets (VI)

The rewrite rule can be applied modulo associativity, commutativity and identity to the distributed state iff its pre-marking is a submultiset of that state.

Furthermore, if the distributed state contains the union of several such presets, then several transitions can fire concurrently.

For example, from `$ $ $` we can get in one concurrent step to `c c a q` by pushing twice (concurrently!) the `buy-c` button and once the `buy-a` button.

Petri Nets (VII)

We can of course ask and get answers to questions about the behaviors possible in this system. For example, if I have a dollar and three quarters, can I get a cake and an apple?

```
Maude> search $ q q q =>+ c a M:Marking .
search in PETRI-MACHINE : $ q q q =>+ c a M:Marking .

Solution 1 (state 4)
states: 5 in 0ms cpu (0ms real)
M:Marking --> null
```

we can also interrogate the search graph,

Petri Nets (VIII)

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ===> state 1 (rl [buy-c]: $ => c .)
arc 1 ===> state 2 (rl [buy-a]: $ => a q .)

state 1, Marking: c q q q

state 2, Marking: a q q q q
arc 0 ===> state 3 (rl [chng]: q q q q => $ .)

state 3, Marking: $ a
arc 0 ===> state 4 (rl [buy-c]: $ => c .)
arc 1 ===> state 5 (rl [buy-a]: $ => a q .)

state 4, Marking: c a

state 5, Marking: a a q
```

Petri Nets (IX)

```
Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl [buy-a]: $ => a q . ]===>
state 2, Marking: a q q q q
===[ rl [chng]: q q q q => $ . ]===>
state 3, Marking: $ a
===[ rl [buy-c]: $ => c . ]===>
state 4, Marking: c a
```

What is Concurrency?

Why was concurrency impossible in our `CANDY-AUTOMATON` example, but possible in our little `PETRI-MACHINE` example?

The problem with `CANDY-AUTOMATON`, and with any LTS having unstructured states, is that its states are atomic, and, having no smaller pieces, cannot be distributed.

By contrast, a Petri net marking is made out of smaller pieces, namely its constituent places, and therefore can be distributed, so that several transitions can happen simultaneously.

What is Concurrency? (II)

Then what, is concurrency about multisets?

Not necessarily; this is the very common fallacy of taking the part for the whole; for example, "Logic Programming = Prolog," or "Concurrency = Petri Nets".

A more fair and open-minded answer is to give the rewriting logic motto:

Concurrent Structure = *Algebraic Structure*.

What is Concurrency? (III)

That is, any algebraic structure in the set of states, other than atomic constants, even a single unary operator, will open the possibility for the states to be distributed, and therefore for transitions being concurrent.

Of course that potential for concurrency may be frustrated by the specific transitions of a system forcing a sequential execution, but the potential is there if we use other transitions.

In summary, there are as many possible styles of concurrent systems as there are signatures Σ and equations E. For example: multiset concurrency, tree concurrency, string concurrency, and many, many other possibilities.

Petri Nets in General

I give the Meseguer-Montanari "Petri nets are monoids" definition, instead than the usual, but less enlightening, multigraph definition.

A place-transition Petri net N consists of:

- a set P of places; we then call markings to the elements in the free commutative monoid $M(P)$ of finite multisets of P.

- a labeled transition system $N = (M(P), L, T)$.

Petri Nets in General (II)

The general transformation associating a rewrite theory $R(N)$ to each Petri net N is then obvious. $R(N)$ has:

- a single sort, named, say $M(P)$, or just *Marking*, with constants the elements of P and a *null* constant.

- a binary operator
  $\_\ \_$: *Marking Marking* $\longrightarrow$ *Marking* $[assoc\ comm\ id:\ null]$

- for each $(m, l, m') \in T$ a rewrite rule $l : m \longrightarrow m'$.