Maude 2.1 Demo

WRLA 2004

Overview

1. New features in 2.1
   - Module expressions.
   - Explicit lightweight polymorphism.
   - upTerm/downTerm functions.
   - More ascent functions.
   - Minor enhancements.
2. Projects using Maude.
   - JavaFAN.
   - Pathway Logic.

Module expressions

- Only allowed for imports.
- Supports arbitrary nesting of renaming and summation.
- Renaming respects import structure - only modules affected by renaming get copied.
- Sorts, operators and labels can be renamed; renamed operators may change their syntactic attributes.
- Modules created for module expressions are "virtual" - only parts of their contents is generated to save memory.
- Caching of module expressions ensures that a given module is only imported once.

Renamed natural numbers

```
fmod 2NAT is
  including NAT + NAT *
   (sort Zero to Zero',
    sort Nat to Nat',
    sort NzNat to NzNat',
    op 0 : -> Zero to z) .
endfm

red 6 * 7 .
red 6 * (7).Nat .
red (6 * 7).Nat' .
red z + 42 .
red 0 + 42 .
```

Explicit lightweight polymorphism

- Introduced to make the lightweight polymorphism implicit in certain builtin operators such as `if_then_else_fi` and `_==_` explicit:

```
op if_then_else_fi : Bool Universal Universal -> Universal
      [poly (2 3 0)
       special (id-hook BranchSymbol
                term-hook 1 (true)
                term-hook 2 (false))] .

op _==_ : Universal Universal -> Bool
      [prec 51 poly (1 2)
       special (id-hook EqualitySymbol
                term-hook equalTerm (true)
                term-hook notEqualTerm (false))] .
```

- However it is available for user defined constructors.

- Instances of polymorphic operators are created on demand on a per kind basis.

Polymorphic lists

```
fmod POLY-LIST is
  sort List .
  op nil : -> List [ctor] .
  op __ : Universal List -> List [ctor poly (1)] .
endfm

fmod POLY-LIST-TEST is
  pr POLY-LIST + CONVERSION + QID .
  op list2string : List -> String .

var L : List . var S : String . var Q : Qid .
var R : Rat . var F : Float .
  eq list2string(nil) = "" .
  eq list2string(S L) = S + list2string(L) .
  eq list2string(Q L) = "'" + string(Q) + list2string(L) .
  eq list2string(R L) = string(R, 10) + list2string(L) .
  eq list2string(F L) = string(F) + list2string(L) .
endfm

red list2string("The answer is " 42.0 " but that is only "
1/2 " of the " 'problem "." nil) .
```

upTerm/downTerm builtin functions

- Move terms from the object level to the metalevel and back.
- Module must include the *object signature* as well as the metalevel signature.
- Typing is solved by the lightweight polymorphism:

```
op upTerm : Universal -> Term
    [poly (1) special (...)] .

op downTerm : Term Universal -> Universal
    [poly (2 0) special (...)] .
```

- Second argument of downTerm provides both the object level kind and the object level return value if the metaterm does not represent a term in that kind.

Remaining ascent functions

- All parts of modules in the module database and whole modules can be lifted to the metalevel:

```
op upModule : Qid Bool ~> Module [...] .
op upImports : Qid ~> ImportList [...] .
op upSorts : Qid Bool ~> SortSet [...] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet
    [...] .
op upOpDecls : Qid Bool ~> OpDeclSet [...] .
```

- Hooks for builtin operators are correctly handled and resulting metamodules can be passed to descent functions.

Up and down example

```
red in META-LEVEL : upModule('META-LEVEL, true) .
red in META-LEVEL : upModule('NAT, true) .

red in META-LEVEL :
downTerm(getTerm(metaReduce(upModule('NAT, true),
                            '_*_[upTerm(6), upTerm(7)])),
         undef:Nat) .

red in META-LEVEL :
downTerm(getTerm(metaReduce(upModule('META-LEVEL, true),
          'metaReduce[upTerm(upModule('NAT, true)),
                      upTerm('_*_[upTerm(6), upTerm(7)])])),
         undef:Term) .
```

Minor enhancements

- Source tree now includes a test suite.
- Standard prelude quo/rem/gcd/lcm/divides functions extended to rationals.
- show module/show all commands now handle specials/hooks.
- Constructor coloring now supported for iter operators.
- Many bug fixes!

JavaFAN (Java Formal ANalyzer)

- Work by Azadeh Farzan, José Meseguer and Grigore Ro cu at Urbana.
- Tool for the formal analysis of Java Virtual Machine (JVM) bytecode produced by compiling Java programs.
- Executable specification of the 150 most commonly used JVM bytecode instructions.
- Deterministic JVM features specified by 300 equations.
- Concurrent JVM features specified by 40 rewrite rules.
- Analysis methods include symbolic execution, safely property checking by proof search and model checking using Maudes LTL model checker.

JavaFAN - Dining Philosophers

- The example uses the Maude LTL model checker to find a deadlock for 4 (naive) philosophers.
- JavaFAN can find a deadlock for version with 9 philosophers.
- It can also prove deadlock freeness for fixed version with 7 philosophers.
- Java PathFinder chokes on 4 philosophers.

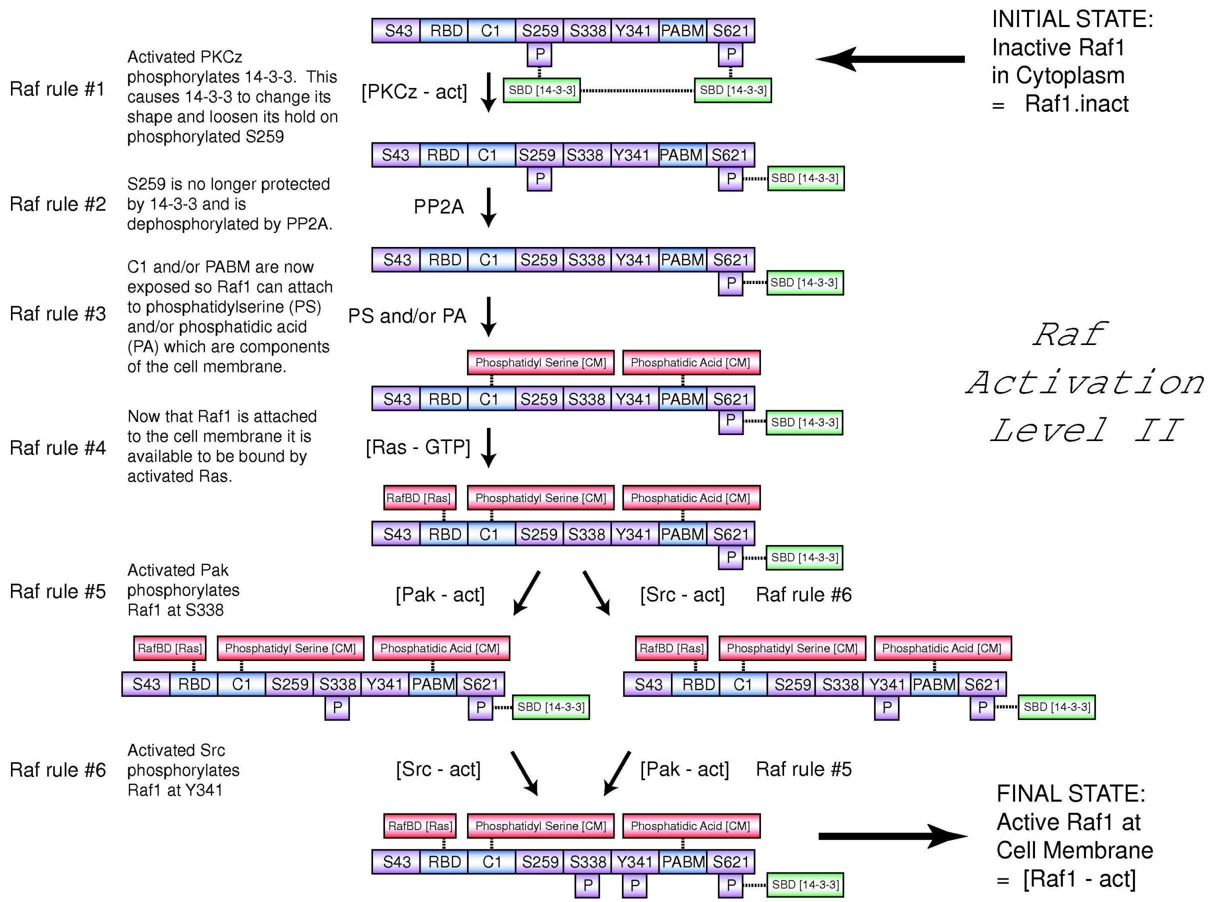
JavaFAN - Thread Game

- Thread Game (J. S. Moore) consists of two process accessing a common variable c.
- Each thread reads c twice and writes the sum of the two values back to c.
- The problem is to find an interleaving of executions that leaves c with a given value n.
- The example uses the Maude seach command to solve this for $n = 50$.

Pathway Logic - Protein Domain Modules

- Proteins are sequences of amino acids; biologists refer to certain subsequences as domains or motifs.
- In the domain level Maude model, a protein is a pair consisting of a name and a multiset of domains (e.g. SBM) and amino acid sites (e.g. (S 43)).
- Both domains and amino acid sites can take modifiers to indicate that they are bound or phosphorylated.
- The overall system state is a multiset of protein and edges between specific domains or amino acid sites.
- Rules encode possible interactions that change peices of the system state.

Pathway Logic - phosphorylation of Raf



Pathway Logic - example rule

```
rl[Raf1#3.PS.PA]:
 {CM | cm PS PA
   {cyto [Raf1 | (S 43), (S 259), (S 338), (Y 341),
                 (S 621 - phos - bound), C1, PABM, raf:Atts]
       [14-3-3a | (SBD - bound), (DMD - bound), 1a:Atts]
       [14-3-3b | SBD, (DMD - bound), (T 141 - phos)]
       e((14-3-3a, DMD), (14-3-3b,DMD))
       e((Raf1, (S 621)), (14-3-3a,SBD))}}
  =>
 {CM | cm PS PA
       [Raf1 | (S 43), (S 259), (S 338), (Y 341),
               (S 621 - phos - bound), (C1 - bound), (PABM - bound), raf:Atts]
       [14-3-3a | (SBD - bound), (DMD - bound), 1a:Atts]
       [14-3-3b | SBD, (DMD - bound), (T 141 - phos)]
       e((14-3-3a,DMD), (14-3-3b,DMD))
       e((Raf1, (S 621)), (14-3-3a,SBD))
       e((Raf1, C1), b(PS)) e((Raf1,PABM), b(PA))
      {cyto}} .
  [ metadata "21278045(R-20)  20379031(D) for PA  99426181(D) for PS"] .
```

Pathway Logic - execution

- findPath runs the model checker and extracts a simple path (list of rule labels and the final state) from the counterexample.
- Look for the desired final state:

```
red findPath(qraf, praf0) .
```

- Look for an expected intermediate state

```
red findPath(qraf, praf1) .
```

- Look for an undesired intermediate state

```
red findPath(qraf, praf2) .
```