Maude 2.2 Demo

WRLA 2006
Overview

• New features in 2.2
  – Random numbers
  – Counters
  – External objects
  – Parameterization
  – Predefined container modules
  – Linear Diophantine solver

• Work in progress
  – Built-in strategy language
Random numbers

• We use the Mersenne Twister as a source of high quality random numbers.

• The sequence of numbers generated from a particular seed is viewed as a function $\text{random}$ from the natural numbers into $[0, \ldots, 2^{32} - 1]$.

• Internally, the state of the Mersenne Twister is cached so that if $\text{random}$ was last called on $n$, calling it on $n + k$ requires $k$ steps of the twister; usually $k = 1$.

• The seed is set at startup time with the command line flag -random-seed=$\langle\text{int}\rangle$ so the sequence is constant for any given run of Maude.
Counters

• It is often useful to have implicitly stored state, especially when working with random numbers.

• Such state cannot be functional but is available in system modules via counters.

• `counter` is a special constant of kind `[Nat]` that each time it is rewritten (by rules) generates the next larger natural number.

• It can be viewed as a special builtin strategy for executing the otherwise nonexecutable rule:

```plaintext
```

• Additional counters can be created using renamed copies. Reseting of counters between commands is avoided by

```plaintext
set clear rules off .
```
### External objects

- Maude 2.2 supports external objects to represent entities in the external world.

- Configurations that want to communicate with external objects must contain at least one portal.
  
  ```
  sort Portal .
  subsort Portal < Configuration .
  op <> : -> Portal [ctor] .
  ```

- `erewrite` is used to start rewrite sequences that may involve external objects.

- `erewrite` may not terminate even if no rewrites are possible; this is because there may be incomplete transactions with external objects and more rewriting may be possible once they complete.
Internet sockets

- Maude 2.2 supports IPv4 TCP client and server sockets.
- Sockets are created by sending a message to a special external object called:

  \[
  \text{op socketManager} : \rightarrow \text{Oid} \ [\text{special} \ (\ldots)] .
  \]
- The created socket external objects are named using the constructor

  \[
  \text{op socket} : \text{Nat} \rightarrow \text{Oid} \ [\text{ctor}] .
  \]
- \textit{send} and \textit{receive} messages can then be sent to the newly created socket objects.
- Message are usually paired - a user object sends a message to an external object and waits for a reply message. This enforces sequentialization in the otherwise concurrent Maude configuration.
Parameterization

- Core Maude now supports a subset of Full Maude module parameterization.

- Modules may now take a list of parameters:
  
  `fmod MAP{X :: TRIV, Y :: TRIV} is`

- Sorts may be parameterized:
  
  `sorts Entry{X,Y} Map{X,Y} .`

- Instances of parameterized modules must be fully instantiated before they can be imported.

- Instantiating by a module-view removes the parameter.

- Instantiation by a theory-view changes the theory of a parameter.

- The final instantiation in a module may instantiate using parameters from the enclosing module.
**Predefined Parameterized Modules 1**

- The predefined module LIST provides associative lists over TRIV.
- Adding or removing elements from either end is amortized constant time except for pathological cases.
- Concatenation and reversal is linear time.
- The predefined module SET provides sets over TRIV.
- Insertion and deletion of elements is $O(\log n)$ time.
- Union, intersection and difference are $O(n \log n)$ time where $n$ is the size of the largest set involved.
- Views from TRIV are provided for all the standard builtin data types.
Predefined Parameterized Modules 2

• The theory TAO-SET describes transitive, antisymmetric orderings (reflexivity is unspecified).

• This structure is just strong enough for merge and sorting to work correctly.

• The predefined module SORTABLE-LIST builds on top of LIST and provides an $O(n \log n)$ time sorting operation for lists over TAO-SET.

• Views from TAO-SET are provided for the standard built-in data types where it makes sense.

• The predefined module LIST-AND-SET provides conversion between lists and sets and allows a list to be filtered by a set.
The predefined module MAP provides maps from TRIV to TRIV with $O(\log n)$ time lookup and updating.

The theory DEFAULT describes data types with a distinguished element.

The predefined module ARRAY closely resembles MAP but has DEFAULT as its target theory.

Mapppings to the distinguished element are not stored, and missing entries are assumed to map to the distinguished element.

Views from DEFAULT are provided for the standard builtin data types where it makes sense.

Nestable versions of lists and sets are provided by LIST* and SET*.
Linear Diophantine solver

- Integer vectors and matrices are implemented as instantiations of module ARRAY.
- The predefined module DIOPHANTINE contains a solver for non-negative solutions of (homogenous and inhomogenous) linear Diophantine equations.
- The solution is a pair of sets of integer vectors $A$ and $B$.
- The non-negative solutions are formed by adding a vector from $A$ to a non-negative linear combination of vectors from $B$.
- Two algorithms are currently implemented - Contejean-Devie and a method based on Gaussian elimination and extended gcd.
Builtin strategy language

- Rewriting with a strategy is invoked by:
  ```
  srewrite [<nat>] in <module> :
  <term> using <strategy> .
  ```

- Leaf strategies implemented so far:
  - `fail` produce the empty set of successors
  - `idle` produce the identity rewrite
  - `all` apply any rule
  - `l` apply a rule with label `l`
  - `l[s]` apply a rule with label `l` and substitution `s`
Built-in strategy language 2

- Combinators implemented so far:
  - \( s * \) do \( s \) 0 or more times
  - \( s + \) do \( s \) 1 or more times
  - \( s ; s' \) do \( s \) and then \( s' \)
  - \( s \mid s' \) do \( s \) or \( s' \)
  - \( t ? s : f \) do \( t \), if it produces results, do \( s \) on them;
    else do \( f \) on the original term