Universal Constructions in Maude

Working Example Module

Throughout we'll use the following module as an example to base our transformations on:

```
fmod MYMOD is
  sorts M N P .
  subsorts N P < M .
  op n : -> N .
  op p : -> P .
  op m : -> M .

  op f_ : M -> M .
  eq f n = p .
  eq f m = m .
  eq f p = p .
endfm
```

Sets in Maude

Using a combination of theories, parameterized modules, and views, we can define sets over the sorts in `MYMOD`. We start with a `TRIV` theory, a parameterized `SET` module (taking a `TRIV` parameter), and views from `TRIV` to each sort in `MYMOD`:

```
fth TRIV is sort Elt . endfth

fmod SET{X :: TRIV} is
  sorts NeSet{X} Set{X} .
  subsort X$Elt < NeSet{X} < Set{X} .

  op mt  : -> Set{X} .

  op __ : NeSet{X} Set{X} -> NeSet{X} [ctor assoc comm id: mt prec 99] .
  op __ : Set{X}   Set{X} -> Set{X}   [ctor ditto] .

  var N : NeSet{X} .
  eq N N = N .
endfm

view M from TRIV to MYMOD is sort Elt to M . endv
view N from TRIV to MYMOD is sort Elt to N . endv
view P from TRIV to MYMOD is sort Elt to P . endv
```

Finally we create a module which extends the parameterized modules with the appropriate views:

```
fmod MYMOD-SET is
  extending SET{M} + SET{N} + SET{P} .
endfm
```

However, this is problematic because we don't have the (often-wanted) mirrored subsort structure. To fix this, the user manually specifies the desired subsorts:

```
fmod MYMOD-SET-SUBSORT is
  extending MYMOD-SET .

  subsort Set{N} < Set{M} .
  subsort Set{P} < Set{M} .
endfm

reduce n p .
```

```
Maude> reduce n p .
Warning: sort declarations for constant mt do not have an unique least sort.
Warning: sort declarations for operator __ failed preregularity check on 10 out of 100 sort tuples. Firs
Warning: sort declarations for associative operator __ are non-associative on 60 out of 1000 sort triple
reduce in MYMOD-SET-SUBSORT : n p .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSet{M}: n p
```

Now, if we want to change the sort structure of MYMOD, we have to change it in two places! Obviously this is not good for maintainability. Additionally, as shown above, we are failing pre-regularity checks. Not shown here are the many advisories Maude gives about `mt` and `_,_` being imported from multiple places.

So we can make a SUBSORT theory, and a SET-SUBSORT parameterized module which is "subsort aware":

```
fth SUBSORT is
  sorts A B .
  subsort A < B .
endfth

fmod SET-SUBSORT{X :: SUBSORT} is
  sorts SetA{X} SetB{X} NeSetA{X} NeSetB{X} .
  subsort X$A < NeSetA{X} < SetA{X} .
  subsort X$B < NeSetB{X} < SetB{X} .
  subsort SetA{X} < SetB{X} .

  op mt  : -> SetA{X} .
  op mt  : -> SetB{X} .

  op __ : NeSetA{X} SetA{X} -> NeSetA{X} [ctor assoc comm id: mt prec 99] .
  op __ : NeSetB{X} SetB{X} -> NeSetB{X} [ctor assoc comm id: mt prec 99] .

  op __ : SetA{X} SetA{X} -> SetA{X} [ctor ditto] .
  op __ : SetB{X} SetB{X} -> SetB{X} [ctor ditto] .

  var NA : NeSetA{X} .
  eq NA NA = NA .

  var NB : NeSetB{X} .
  eq NB NB = NB .
endfm
```

```
view N<M from SUBSORT to MYMOD is sort A to N . sort B to M . endv
view P<M from SUBSORT to MYMOD is sort A to P . sort B to M . endv

fmod MYMOD-SET-SUBSORT-2 is
  extending SET-SUBSORT{N<M} + SET-SUBSORT{P<M} .
endfm

reduce n p .
```

```
Warning: sort declarations for constant mt do not have an unique least sort.
Warning: sort declarations for operator __ failed preregularity check on 17 out of 144 sort tuples. Firs
Warning: sort declarations for associative operator __ are non-associative on 90 out of 1728 sort triple
==========================================
reduce in MYMOD-SET-SUBSORT-2 : n p p .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSetB{P<M}: n p
```

This fails pre-regularity checks as well, and has a slightly more convoluted sort as output. Additionally, if we had a module with *no* subsorts (eg. a many-sorted module), we would *not* be able to use this approach to construct sets, because there would be no view to `SUBSORT` possible.

What do we Actually Want?

Think of this as a universal construction. For every part of a specified theory, we want to guarantee the existence of another part. For example, for every sort `X` in a theory, we want to guarantee that `Set{X}` and `NeSet{X}` exist. Additionally, for every subsort `A < B`, we want to make sure that `Set{A} < Set{B}` and `NeSet{A} < NeSet{B}`.

This universal module has a "commutative diagram" flavor, where certain parts of the theory are called out as already existing (the `forall`), and as a result we ensure that other parts exsit (the `exists`).

```
univ SET is

  --- Maude's normal notion of `SET{A}`, for `A` a view to `TRIV`
  forall:
    sort A .
  exists:
    sorts NeSet{$A} Set{$A} .
    subsort $A < NeSet{$A} < Set{$A} .

    op mt : -> Set{$A} .
    op __ : Set{$A}   Set{$A} -> Set{$A}   [ctor assoc comm id: mt prec 99] .
    op __ : NeSet{$A} Set{$A} -> NeSet{$A} [ctor ditto] .

    var NA : NeSet{$A} .
    eq NA NA = NA .

  --- automatic subsort generation over new sorts
  forall:
    sorts A B NeSet{$A} NeSet{$B} Set{$A} Set{$B} .
    subsort $A < $B .
  exists:
    subsort NeSet{$A} < NeSet{$B} .
    subsorts Set{$A} < Set{$B} .
  --- automatically lift each operator on sort `A` to work on sort `Set{A}`
  --- denote that C D are strings of sorts (possibly \eps) with `sorts*`
  --- only works with SET because order doesn't matter
  forall:
    sorts A B Set{$A} Set{$B} NeSet{$A} .
    sorts* C D .
    op f : $C $A $D -> $B .
  exists:
    op $f : $C Set{$A} $C' -> Set{$B} .

    var a : $A . vars NA NA' : NeSet{$A} .
    var cs : $C . var ds : $D .
    eq $f(cs, mt, ds)       = mt .
    eq $f(cs, (NA NA'), ds) = $f(cs, NA, ds) $f(cs, NA', ds) .

enduniv
```

Semantics

Here we'll give "semantics by example", as actual semantics aren't fleshed out. Two options are provided: (i) desugaring into Maude's theories, parameterized-modules, and views; (ii) syntactic transformation on the original definition.

As Theories/Parameterized-Modules/Views

First we apply the transformation described by the first part of the universal construction. Notice that the universal part (`forall: ...`) corresponds to a functional theory, and the existential part (`exists: ...`) corresponds to a parameterized-module (as you would expect given the initial/free semantics respectively).

```
fth SET-THEORY-1 is
  sort A .
endfth

fmod SET-MODULE-1{X :: SET-THEORY-1} is
  sorts Set{X} NeSet{X} .
  subsort X$A < NeSet{X} < Set{X} .
  op mt : -> Set{X} .
  op __ : Set{X} Set{X} -> Set{X} [ctor assoc comm id: mt prec 99] .
  op __ : Set{X} NeSet{X} -> NeSet{X} [ctor ditto] .
  var a : NeSet{X} .
  eq a a = a .
endfm
```

Intermediate views and a module are generated:

```
view SET-1-M from SET-THEORY-1 to MYMOD is sort A to M . endv
view SET-1-N from SET-THEORY-1 to MYMOD is sort A to N . endv
view SET-1-P from SET-THEORY-1 to MYMOD is sort A to P . endv

fmod MYMOD-EXTENDED-1 is
  protecting MYMOD .
  protecting   SET-MODULE-1{SET-1-M}
             + SET-MODULE-1{SET-1-N}
             + SET-MODULE-1{SET-1-P} .
endfm
```

Then the next theory is generated:

```
fth SET-THEORY-2 is
  sorts A B Set{A} Set{B} NeSet{A} NeSet{B} .
  subsort A < B .
endfth

fmod SET-MODULE-2{X :: SET-THEORY-2} is
  subsort X$Set{A} < X$Set{B} .
  subsort X$NeSet{A} < X$NeSet{B} .
endfm
```

And finally we generate the appropriate views (from the second theory into the intermediate generated module), along with the final module:

```
view MYMOD-N-M from SET-THEORY-2 to MYMOD-EXTENDED-1 is
  sort A to N . sort Set{A} to Set{SET-1-N} . sort NeSet{A} to NeSet{SET-1-N} .
  sort B to M . sort Set{B} to Set{SET-1-M} . sort NeSet{B} to NeSet{SET-1-M} .
endv

view MYMOD-P-M from SET-THEORY-2 to MYMOD-EXTENDED-1 is
  sort A to P . sort Set{A} to Set{SET-1-P} . sort NeSet{A} to NeSet{SET-1-P} .
  sort B to M . sort Set{B} to Set{SET-1-M} . sort NeSet{B} to NeSet{SET-1-M} .
endv

fmod MYMOD-EXTENDED is
  protecting MYMOD-EXTENDED-1 .
  protecting   SET-MODULE-2{MYMOD-N-M}
             + SET-MODULE-2{MYMOD-P-M} .
endfm

reduce mt p n m p .
```

So we see that the universal clause `forall` represents the *loose* semantics of a functional theory, and the `exists` clause represents the *free* semantics over the variables declared in the preceding theory.

Unfortunately, the resulting module `MYMOD-EXTENDED` has strange names for everything that has been generated, for example:

```
Maude> reduce mt p n m p .
reduce in MYMOD-EXTENDED : p,n,p,m .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Set{SET-1-M}: n p m
```

Syntactic Transformation

Here we see that `n , p , m` has sort `Set{SET-1-M}`, where we would actually like it to have sort `Set{M}` for readability and ease-of-use. It's not clear that simpler names could easily be generated, especially when intermediate modules/views/theories are needed. If instead the semantics are given in terms of a definition transformation, the whole issue of cluttering name-spaces is avoided. Intuitively, the module `MYMOD` is extended to:

```
fmod MYMOD-EXTENDED-CLEAN is
  sorts M N P .
```

```
subsorts N P < M .
op n : -> N .
op p : -> P .
op m : -> M .

op f_ : M -> M .
eq f n = p .
eq f m = m .
eq f p = p .

--- first construction with { A |-> M }
sorts NeSet{M} Set{M} .
subsort M < NeSet{M} < Set{M} .

op mt  : -> Set{M} .
op __ : Set{M}   Set{M} -> Set{M}   [ctor assoc comm id: mt prec 99] .
op __ : NeSet{M} Set{M} -> NeSet{M} [ctor ditto] .

var NA1 : NeSet{M} .
eq NA1 NA1 = NA1 .

--- first construction with { A |-> N }
sorts NeSet{N} Set{N} .
subsort N < NeSet{N} < Set{N} .

op mt  : -> Set{N} .
op __ : Set{N}   Set{N} -> Set{N}   [ctor assoc comm id: mt prec 99] .
op __ : NeSet{N} Set{N} -> NeSet{N} [ctor ditto] .

var NA2 : NeSet{N} .
eq NA2 NA2 = NA2 .

--- first construction with { A |-> P }
sorts NeSet{P} Set{P} .
subsort P < NeSet{P} < Set{P} .

op mt  : -> Set{P} .
op __ : Set{P}   Set{P} -> Set{P}   [ctor assoc comm id: mt prec 99] .
op __ : NeSet{P} Set{P} -> NeSet{P} [ctor ditto] .

var NA3 : NeSet{P} .
eq NA3 NA3 = NA3 .

--- second construction with { A |-> N , B |-> M }
subsort Set{N} < Set{M} .
subsort NeSet{N} < NeSet{M} .

--- second construction with { A |-> P , B |-> M }
subsort Set{P} < Set{M} .
subsort NeSet{P} < NeSet{M} .

--- third construction with { A |-> M , B |-> M , f |-> f_ }
op f_ : Set{M} -> Set{M} .
```

```
  var a : M . vars NA NA' : Set{M} .
  eq f mt       = mt .
  eq f (NA NA') = (f NA) (f NA') .
endfm

reduce mt p n m p .
reduce f (mt p n m p) .
```

This eliminates many of the advisories Maude gives about operators being imported from many places, and has nicer names for all the associated sorts:

```
reduce in MYMOD-EXTENDED-CLEAN : p n p m .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSet{M}: n p m

reduce in MYMOD-EXTENDED-CLEAN : f (p n p m) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result NeSet{M}: p m
```

Where it Breaks Down

Some views are more involved (eg. sending theory operators to derived terms), or may have complicated proof obligations (which are not syntactically checkable or automatically dischageable). An anonymous view is not suitable in this case. Take a `POSET` for example:

```
fth POSET is
  sort Elt .

  op _<_ : Elt Elt -> Bool .
  op _<=_ : Elt Elt -> Bool .

  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z   [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X          [nonexec label antisymmetric] .
  eq X <= X = true                     [nonexec] .
  ceq X <= Y = true if X < Y           [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

It may be difficult for Maude to automatically find every possible `view` of `POSET` in a given module, especially since the `view` can send the theory operators to derived terms. But, we can still gain in *extensibility* even using these theories:

```
univ LEX-PAIR is

  forall:
    view X from POSET .
    view Y from POSET .
  exists:
    sort Pair{$X,$Y} .
    op <_;_> : $(X.Elt) $(Y.Elt) -> Pair{$X,$Y} .
    op _<_ : Pair{$X,$Y} Pair{$X,$Y} -> Bool .
    op 1st : Pair{$X,$Y} -> $(X.Elt) .
    op 2nd : Pair{$X,$Y} -> $(Y.Elt) .
    vars A A' : $(X.Elt) .
    vars B B’ : $(Y.Elt) .
    eq 1st(< A ; B >) = A .
    eq 2nd(< A ; B >) = B .
    eq < A ; B > < < A’ ; B’ > = (A < A’) or (A == A’ and B < B’) .

enduniv
```

While the hard work of demonstrating a `view` to `POSET` is left to the user, at least the instantiation of two `POSET`s into a single `LEX-PAIR` is automatic.

If Then Else

Instead of having `if_then_else_fi` be “magic” using `poly` in Maude, we can explicity construct an `if_then_else_if` for each kind.

```
univ CONDITIONAL is
  protecting BOOL .

  forall:
    sort A .
  exists:
    op if_then_else_fi : Bool $[A] $[A] -> $[A] .
    vars a1 a2 : $[A] .
    eq if true  then a1 else a2 fi = a1 .
    eq if false then a1 else a2 fi = a2 .

enduniv
```

Symbolic Terms

We want to be able to put variables anywhere into our terms, so for each sort `A` we declare a subsort `Var{A}`. We also complete the `Var` heirarchy with a bottom element, so that if you don’t care what sort of variable you’re using you can use that sort.

```
univ VAR is

  forall:
    sort A .
  exists:
    sorts Var{$A} Var{$[A]} .
    subsorts Var{$[A]} < Var{$A} < $A .

  forall:
    sorts A B Var{$A} Var{$B} .
    subsort $A < $B .
  exists:
    subsort Var{$A} < Var{$B} .

enduniv
```

Now you might want to be able to distinguish at the sort level that a term is ground (does not contain variables). To do so, we make a subsort `Ground{A}` of each sort `A`, with all the operators copied down to operate over `Ground{A}` as well. This allows equational simplification to happen over terms and terms with variables, but allows sort-level distinction of terms without variables.

```
univ GROUND is

  forall:
    sort A .
  exists:
    sort Ground{$A} .
    subsort Ground{$A} < $A .

  forall:
    sorts A B Ground{$A} Ground{$B} .
    subsort A < B .
  exists:
    subsort Ground{$A} < Ground{$B} .

  forall:
    sort* A .
    sort B .
    op f : $A -> $B .
  exists:
    op f : Ground{$A} -> Ground{$B} .

enduniv
```

Finally, you may want to be able to perform substitutions of variables for terms. To do so, we define the substitution homomorphism over all terms of sort `A` which may have variables in them. This requires that `VAR` and `CONDITIONAL` are defined over the sort-heirarchy of interest.

```
univ SUBSTITUTION is
  using VAR + CONDITIONAL .

  forall:
    sorts A Var{$A} .
  exists:
    sort Subst{$A} .
    op _:=_ : Var{$A} $A -> Subst{$A} .
    op _[_] : $A Subst{$A} -> $A .
    vars va1 va2 : Var{$A} . var a : $A .
    eq va1 [va2 := a] = if va1 == va2 then a else va1 fi .

  forall:
    sorts A B Subst{$A} Subst{$B} .
    subsort A < B .
  exists:
    subsort Subst{$A} < Subst{$B} .

  forall:
    sort* A .
    sorts B C Subst{$C} .
    op f : $A -> $B .
  exists:
    op _[_] : $B Subst{$C} -> $B .
    var as : $A . var sc : Subst{$C} .
    eq $f(as)[sc] = $f(as[sc]) .
```

```
enduniv
```

Functions

Here we define functions between one sort heirarchy any another, including appropriate sub-sorting relations (contravariant on domain, covariant on range).

```
univ FUNCTION is
  using SUBSTITUTION .

  --- function sorts
  forall:
    sorts A B .
  exists:
    sort $A=>$B .
    op __ : $A=>$B $A -> [$B] .

  forall:
    sorts A B C .
    subsort $A < $B .
  exists:
    subsort $C=>$A < $C=>$B .
    subsort $B=>$C < $A=>$C .

  forall:
    sort A .
  exists:
    op id : -> $A=>$A .
    var a : $A .
    eq id a = a .

  forall:
    sorts A B C .
  exists:
    op _._ : $B=>$C $A=>$B -> $A=>$C .
    var f : $B=>$C . var g : $A=>$B . var A : $A .
    eq id . g = g .
    eq f . id = f .
    eq (f . g) A = f(g(A)) .

  --- lambda abstraction
  --- using Ground{$A} to avoid variable capture
  --- eventually would like to lift this restriction
  forall:
    sorts A B .
    sorts Var{$A} Subst{$A} Ground{$A} $A=>$B .
  exists:
    op \_._ : Var{$A} $B -> $A=>$B .
    var ga : Ground{$A} . var va : Var{$A} . var b : $B .
    eq (\ va . b) ga = b [va := ga] .

enduniv
```

Here we define an explicit `map` function for sets instead of generating an implicit operator over sets for ever operator over the base sorts.

```
univ MAPPABLE-SET is
  using SET + FUNCTION .

  forall:
    sorts A B .
  exists:
    op map__ : $A=>$B Set{$A} -> Set{$B} .

    var f : $A=>$B . var A : $A . vars NA NA' : NeSet{$A} .

    eq map f A          = f A .
    eq map f mt         = mt .
    eq map f (NA , NA') = map f NA , map f NA' .

enduniv
```