Meta-Programming with Modelica

for Meta-Modeling and Model Transformations

Peter Fritzson, Adrian Pop
OpenModelica Course, 2007 02 05

Extensibility and Modularity of Modeling Tools

• Modeling and simulation tools are too monolithic
• Models and tools need to be extensible and modular
• Creation, query, manipulation, composition and management of models
• Need for modeling of operations on models
Some Semantics Modeling Approaches

- Extensibility - allow models to also model language properties
- Ontologies to classify application domains
  - For example, semantic web
- Equation-based approaches
  - Modelica – hybrid differential algebraic equations
  - Single-assignment equations – functional languages, SOS/Natural semantics
  - Unification equations: logic programming and functional languages, SOS/Natural semantics
  - Can usually be efficiently executed
- Logic approaches
  - First-order logic
  - Often computationally intractable for realistic applications in its general form

Meta-Level Operations on Models

- Model Operations
  - Creation
  - Query
  - Manipulation,
  - Composition
  - management
- Manipulation of model equations for
  - Optimization purposes
  - Parallelization
  - Model checking
  - Simulation with different solvers
- Modularity
  - Allow model packages for tool extensions
  - Example: Automatic PDE discretization schemes
Meta-Level Operations on Models, Cont.

- Model configuration for simulation purposes
  - Initial state
  - Initialization via xml data files or databases
- Simulation features
  - Running a simulation and display a result
  - Running more simulations in parallel
  - Handle simulation failures and continue the simulation in a different way
  - Possibility to generate ONLY specific data from a simulation
  - Possibility to manipulate simulation data for export to another tool

Meta-Modelica Compiler (MMC) and Language

- Supports extended *subset* of Modelica
- Used for development of OMC
- Some MetaModelica Language properties:
  - Modelica syntax and base semantics
  - Pattern matching (named/positional)
  - Local equations (local within expression)
  - Recursive tree data structures
  - Lists and tuples
  - Garbage collection of heap-allocated data
  - Arrays (with local update as in standard Modelica)
  - Polymorphic functions
  - Function formal parameters to functions
  - Simple builtin exception (failure) handling mechanism
A Simple match-Expression Example

- Example, returning a number, given a string

```plaintext
String s;
Real x;
algorithm
x :=
  match continue s
  case "one" then 1;
  case "two" then 2;
  case "three" then 3;
  else
  end match continue;
```

Tree Types – uniontype Declaration Example

- Union types specifies a union of one or more record types
- Union types can be recursive - can reference themselves
- The Exp type is a union type of three record types
- Record constructors INT, NEG and ADD
- Common usage is abstract syntax trees.

MetaModelica tree type declaration:

```plaintext
uniontype Exp
  record INT Integer x1; end INT;
  record NEG Exp x1; end NEG;
  record ADD Exp x1; Exp x2; end ADD;
end Exp;
```

![ADD tree diagram]
Another uniontype Declaration of Exp Expressions

Abstract syntax tree data type declaration of Exp:

```
uniontype Exp
  record INTconst Integer x1; end INTconst;
  record PLUSop Exp x1; Exp x2; end PLUSop;
  record SUBop Exp x1; Exp x2; end SUBop;
  record MULop Exp x1; Exp x2; end MULop;
  record DIVop Exp x1; Exp x2; end DIVop;
  record NEGop Exp x1; end NEGop;
end Exp;
```

```
12+5*13

PLUSop(INTconst(12), MULop(INTconst(5), INTconst(13)))
```

Simple Expression Interpreter – with equation keyword, match, case

```
function eval "Evaluates integer expression trees"
  input Exp exp;
  output Integer intval;
algorithm
  intval :=
    match continue exp
      local INTconst v1, v2; Exp e1, e2;
      case INTconst(v1) then v1;
      case PLUSop(e1, e2) equation
        v1 = eval(e1); v2 = eval(e2); then v1+v2;
      case MULop(e1, e2) equation
        v1 = eval(e1); v2 = eval(e2); then v1*v2;
      case SUBop(e1, e2) equation
        v1 = eval(e1); v2 = eval(e2); then v1-v2;
      case DIVop(e1, e2) equation
        v1 = eval(e1); v2 = eval(e2); then v1/v2;
      case NEGop(e1) equation
        eval(e1) = v1; then -v1;
    end match continue;
end eval;
```
Example: Simple Symbolic Differentiator

```
function difft "Symbolic differentiation of expression with respect to time"
input Exp expr;
input list<IDENT> timevars;
output Exp diffexpr;
algorithmdiffexpr :=
matchcontinue (expr, timevars)
local Exp e1prim,e2prim,tvars;
Exp e1,e2,id;
// der of constant
case (RCONST(_),_) then RCONST(0.0);
// der of time variable
case IDENT("time"),_ then RCONST(1.0);
// der of any variable id
case [id as IDENT(_,tvars)] then
  if listMember(id,tvars) then
    CALL(IDENT("der"),list(id))
  else
    RCONST(0.0);
...
// (e1+e2)' => e1'+e2'
case (ADD(e1,e2),tvars) equation
  e1prim = difft(e1,tvars);
  e2prim = difft(e2,tvars);
  then ADD(e1prim,e2prim);
// (e1-e2)' => e1'-e2'
case (SUB(e1,e2),tvars) equation
  e1prim = difft(e1,tvars);
  e2prim = difft(e2,tvars);
  then SUB(e1prim,e2prim);
// (e1*e2)' => e1'*e2 + e1*e2'
case (MUL(e1,e2),tvars) equation
  e1prim = difft(e1,tvars);
  e2prim = difft(e2,tvars);
  then PLUS(MUL(e1prim,e2),
            MUL(e1,e2prim));
  ...
```

General Syntactic Structure of match-expressions

```
matchcontinue <expr> <opt-local-decl>

  case <pat-expr> <opt-local-decl>
    <opt-equations>
    then <expr>;
  case <pat-expr> <opt-local-decl>
    <opt-equations>
    then <expr>;
  ...
  else <opt-local-decl>
    <opt-equations>
    then <expr>;
end matchcontinue;
```
Semantics of Local Equations in match-Expressions

- Only algebraic equations are allowed, no differential equations.
- Only locally declared variables (local unknowns) declared by local declarations within the case expression are solved for.
- Equations are solved in the order they are declared. (This restriction may be removed in the future).
- If an equation or an expression in a case-branch of a matchcontinue-expression fails, all local variables become unbound, and matching continues with the next branch.

Semantics of Local Equations cont...

- Certain equations in match-expressions do not solve for any variables – they may be called "constraints"
  - All variables are already bound in these equations
  - The equation may either be fulfilled (succeed) or not (fail)
  - Example:

```modelica
local
  Real x=5; Integer y = 10;

equation
  true = x>4; // Succeeds!
  true = y<10; // Fails!!
```

- Thus, there can locally be more equations than unbound variables, if including the constraints.
List Data Structures

- **list** – `list<type-expr>` is a list type constructor
  - Example: `type Reallist = list<Real>;` type is a list of reals
  - Example: `list<Real> rlist;` variable that is a list of reals
- **list** – `list(el1,el2,el3,...)` is a list data constructor that creates a list of elements of identical type.
  - `{}` or `list()` empty list
  - `(2,3,4)` or `list(2,3,4)` list of integers
  - Allow `{el1,el2,el3,...}` overloaded array or list constructor, interpreted as `array(...)` or `list(...)` depending on type context.
- `{}` or `list()` denotes an empty reference to a list or tree.
- `cons` – `cons(element, lst)` adds an element in front of the list `lst` and returns the resulting list.
- Also as `::` operator – `element::lst`

Predefined Polymorphic List Operations

```plaintext
function listAppend
    input list<Eltype> lst1;
    input list<Eltype> lst2;
    output list<Eltype> lst3;
    replaceable type Eltype subtypeof Any;
    end listAppend;

function listReverse
    input list<Eltype> lst1;
    output list<Eltype> lst3;
    replaceable type Eltype subtypeof Any;
    end listReverse;

function listLength
    input list<Eltype> lst1;
    output Integer len;
    replaceable type Eltype subtypeof Any;
    end listLength;

function listDelete
    input ListType lst1;
    input Integer elindex;
    output ListType lst3;
    replaceable type Eltype subtypeof Any;
    type ListType = list<Eltype>;
    end listDelete;
```

function listMember
    input Eltype elem;
    input list<Eltype> lst2;
    output Boolean result;
    replaceable type Eltype subtypeof Any;
    end listMember;

function listNth
    input list<Eltype> lst1;
    input Integer elindex;
    output Eltype elem;
    replaceable type Eltype subtypeof Any;
    end listNth;
```
Function Formal Parameters

- Functions can be passed as actual arguments at function calls.
- Type checking done on the function formal parameter type signature, not including the actual names of inputs and outputs to the passed function.

```
function intListMap   "Map over a list of integers"
  input Functype func;
  input list<Integer> inlst;
  output list<Integer> outlst;
public
  partial function Functype input Integer x1; output Integer x2; end Functype;
algorithm ...
end intListMap;
```

```
function listMap   "Map over a list of elements of Type_a, a type parameter"
  input Functype func;
  input list<Type_a> inlst;
  output list<Type_a> outlst;
public
  replaceable type Type_a;
  partial function Functype input Type_a x1; output Type_a x2; end Functype;
algorithm ...
end listMap;
```

Calling Functions with Function Formal Parameters and/or Parameterized Types

- Call with passed function arguments: `int_list_map(add1,intlst1)`
  Declared using type Int
- Compiler uses type inference to derive type of replaceable type parameter `Type_a = Integer`
  from input list type `list<Integer>` in `listMap(add1, intlst1);

```
// call function intListMap   "Map over a list of integers"
list<Integer> intlst1 := {1,3,5,9};
intlst2 := intListMap(add1, intlst1);
```

```
// call function listMap   "Map over a list of Type_a - a type parameter"
list<Integer> intlst1 := {1,3,5,9};
list<Integer> intlst2;
intlst2 := listMap(add1, intlst1); // The type parameter is
```
Tuple Data Structures

- Tuples are anonymous records without field names
- `tuple<...>` – tuple type constructor (keyword not needed)
  - Example: `type VarBND = tuple<Ident, Integer>;`
  - Example: `tuple<Real,Integer> realintpair;`
- `(..., ..., ...)` – tuple data constructor
  - Example: `(3.14, "this is a string")`
- Modelica functions with multiple results return tuples
  - Example: `(a,b,c) := foo(x, 2, 3, 5);`

Option Type Constructor

- The `Option` type constructor, parameterized by some type (e.g., `Type_a`) creates a kind of uniontype with the predefined constructors `NONE()` and `SOME(...)`:
  ```
  replaceable type Type_a subtypeof Any;
  type Option_a = Option<Type_a>;
  ```
  - The constant `NONE()` with no arguments automatically belongs to any option type. A constructor call such as `SOME(x1)` where `x1` has the type `Type_a`, has the type `Option<Type_a>`.
  - Roughly equivalent to:
    ```
    uniontype Option
      record NONE end NONE;
      record SOME Type_a x1; end SOME;
      replaceable type Type_a subtypeof Any;
      end Option;
    ```
Testing for Failure

- A local equation may fail or succeed.
- A builtin equation operator `failure(arg)` succeeds if `arg` fails, where `arg` is a local equation

Example, testing for failure in Modelica:

```modelica
case ((id2,_) :: rest, id)
  equation
    failure(true = id == id2); value = lookup(rest, id);
  then value;
```

Generating a fail "Exception"

- A call to `fail()` will fail the current case-branch in the match-expression and continue with the next branch.
- If there are no more case-branches, the enclosing function will fail.
- An expression or equation may fail for other reasons, e.g. division by zero, no match, unsolvable equation, etc.
as-expressions in Pattern Expressions

- An unbound local variable (declared local) can be set equal to an expression in a pattern expression through an as-expression (\texttt{var as subexpr})
- This is used to give another name to \texttt{subexpr}
- The same variable may only be associated with one expression
- The value of the expression equation (\texttt{var as subexpr}) is \texttt{subexpr}
- Example:
  - \texttt{(a as Absyn.IDENT("der"), expl,b,c)}

Summary of New MetaModelica Reserved Words

- \texttt{ underscore} is a reserved word used as a pattern placeholder, name placeholder in anonymous functions, types, classes, etc.
- \texttt{(match} is used in match-expressions).
- \texttt{matchcontinue} is used in matchcontinue-expressions.
- \texttt{case} is used in match/matchcontinue-expressions.
- \texttt{local} is used for local declarations in match expressions, etc.
- \texttt{uniontype} for union type declarations, e.g. tree data types.
- \texttt{as} for as-expressions
### Summary of New Reserved Words Cont'

- `list` can be a reserved word, but this is not necessary since it is only used in `list(...)` expressions
- `Option` is a predefined parameterized union type

### Summary of New Builtin Functions and Operators

- `list<...>` – list type constructor, in type context
- `tuple<...>` – tuple type constructor
- `list(...)` – list data constructor, in expression context
- `cons(element, lst)` – attach `element` at front of list `lst`
- `fail()` – Raise fail exception, having null value
- `(..., ..., ...) or tuple(..., ..., ...)` – tuple data constructor
- `::` – List `cons` operator
Conclusions

- Meta-modeling increasingly important, also for the Modelica language and its applications
- Meta-modeling/meta-programming extensions allow writing a Modelica compiler in Modelica
- Extensions are recursive union types (trees), lists, and match-expressions – standard constructs found in functional languages
- OpenModelica compiler implemented using MetaModelica extensions since March 2006.