Meta-Programming with Modelica

for MetaModeling and Model Transformations

Peter Fritzson, Adrian Pop, Martin Sjölund
OpenModelica Course, 2010-11-08

Peter Fritzson

MODELICA pelab

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**

Peter Fritzson

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**

Peter Fritzson

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**

MODELICA   pelab

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**

Peter Fritzson

MODELICA pelab

Meta-Modelica Compiler (MMC) and Language

- **MetaModelica is an extended *subset* of Modelica**
    - **RML compiler supports only MetaModelica**
    - **OMC supports Modelica 3.1 + MetaModelica**
- **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 (different from standard Modelica)**
    - **Vectors (array with local update as in standard Modelica)**
    - **Polymorphic functions**
    - **Function formal parameters to functions**
    - **Simple builtin exception (failure) handling mechanism**

Peter Fritzson

A Simple match-Expression Example

- **Example, returning a number, given a string**
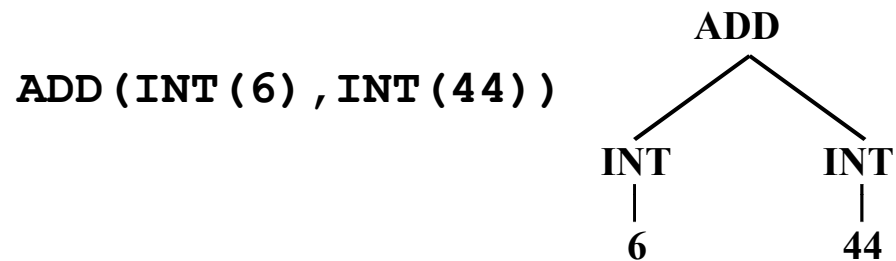
```
String s;
Real   x;
algorithm
x :=
  matchcontinue s
    case "one"   then 1;
    case "two"   then 2;
    case "three" then 3;
    else              0;
  end matchcontinue;
```

Peter Fritzson

MODELICA   pelab

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.**
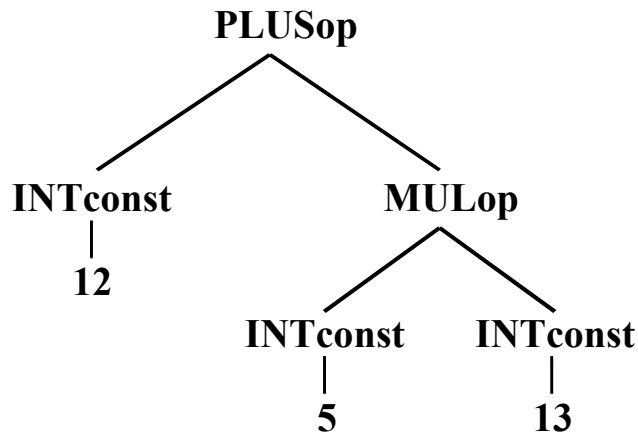
MetaModelica tree type declaration:

```
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(INT(6),INT(44))
```

Another uniontype Declaration of Exp Expressions

Abstract syntax tree data type declaration of Exp:

```
               PLUSop

INTconst              MULop

    12          INTconst    INTconst

                  5            13
```

```
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)))
```

MODELICA   pelab

Simple Expression Interpreter – with equation keyword, match, case

```
function eval "Evaluates integer expression trees"
  input  Exp      exp;
  output Integer intval;
algorithm
 intval :=
  matchcontinue exp
    local Integer 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 SUBop(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 DIVop(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2); then v1/v2;
    case NEGop(e1) equation
      eval(e1) = v1; then  -v1;
  end matchcontinue;
end eval;
```

Local variables with scope inside case expression

Pattern binding local pattern variables e1, e2

Local equations with local unknowns v1, v2

A returned value

Peter Fritzson

MODELICA   pelab

Transformation Example
Simple Symbolic Differentiator

```
protected function diff
  input Exp expr;
  input String timevar;
  output Exp diffExpr;
algorithm
diffExpr :=
matchcontinue (expr,timevar)
  local
    String id,id1,id2;
    Exp e1prim,e2prim,e1,e2;
  // der of constant
  case (INT(_), _) then INT(0);
  // der of time variable
  case(IDENT(id1), id2)
    equation
      true = id1 ==& id2;
    then INT(1);
  // der of time-independent variable
  case (IDENT(_), _) then INT(0);
  ...
```

```
// (e1+e2)' => e1'+e2'
case (ADD(e1,e2),id)
  equation
    e1prim = diff(e1,id);
    e2prim = diff(e2,id);
  then ADD(e1prim,e2prim);
// (e1/e2)' => (e1'*e2 - e1*e2')/e2*e2
case (DIV(e1,e2),id)
  equation
    e1prim = diff(e1,id);
    e2prim = diff(e2,id);
  then DIV(
          SUB(MUL(e1prim,e2),
              MUL(e1,e2prim)),
          MUL(e2,e2));
// (-e1)' => -e1'
case (NEG(e1),id)
  equation
    e1prim = diff(e1,id);
  then NEG(e1prim);
...
```

General Syntactic Structure of match-expressions

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

  case <pat-expr>
    <opt-equations>
    then <expr>;
  case <pat-expr>
    <opt-equations>
    then <expr>;
  ...
  else
    <opt-equations>
    then <expr>;

end matchcontinue;
```

MODELICA   pelab

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.**

Peter Fritzson

MODELICA pelab

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:**

```
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**

MODELICA pelab

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***

MODELICA   pelab

What Does PolyMorphic Mean?

- **PolyMorphic – adapt to multiple types**

- **Poly – multiple, morphic – adapt**

- **Standard – An operation is only defined for one type, e.g. `arrayInsertElement(intelement, intArr)`**

- **A polymorphic function is defined for multiple types, e.g. elements in an array can be of any type: `arrayInsertElement(anyElement, anyArr)`**

Peter Fritzson

MODELICA   pelab

Predefined Polymorphic List Operations

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

MODELICA   pelab

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 o 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;
```

MODELICA   pelab

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};
list<Integer> intlst2;

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 inferred
```

MODELICA pelab

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);`

MODELICA pelab

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;
```

MODELICA   pelab

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:

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

MODELICA pelab

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.

MODELICA   pelab

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-expresson (`var as subexpr`)**

- **This is used to give another name to `subexpr`**

- **The same variable may only be associated with one expression**

- **The value of the expression equation (`var as subexpr`) is `subexpr`**

- **Example:**

  - **`(a as Absyn.IDENT("der"), expl,b,c)`**

MODELICA pelab

match and matchcontinue

- **MetaModelica has two different match-expressions**
    - **matchcontinue runs all matching cases in order until it finds one that succeeds**
    - **match runs only the first matching case**
    - **both are useful to describe different program flows**

```
function notOne "Fails if the input is not 1"
  input  Integer i;
algorithm
  _ := match i
    case 1 then fail();
    // Do stuff
  end match;
end notOne;
```

```
function notOne "Fails if the input is not 1"
  input  Integer i;
algorithm
  _ := matchcontinue i
    case i
      equation
// This constraint needs to be added to the
// beginning of every subsequent case
        false = i == 1;
        // Do stuff
  end matchcontinue;
end notOne;
```

MODELICA   pelab

Summary of New MetaModelica Reserved Words

- **_ Underscore is a reserved word used as a pattern placeholder, name placeholder in anonymous functions, types, classes, etc.**

- **`match` is used in match-expressions.**

- **`matchcontinue` is used in matchcontinue-expressions.**

- **`case` is used in match/matchcontinue-expressions.**

- **`local` is used for local declarations in match expressions, etc.**

- **`uniontype` for union type declarations, e.g. tree data types.**

- **as for as-expressions**

Peter Fritzson

MODELICA pelab

Summary of New Reserved Words Cont'

- `list` **could be a reserved word, but this is not necessary since it is only used in** `list(...)` **expressions**

- `Option` **is a predefined parameterized union type**

Peter Fritzson

MODELICA pelab

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

MODELICA pelab

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.**

- **Bootstrapping - Ongoing work to compile OpenModelica compiler using itself, completed winter 2010 - spring 2011**

Peter Fritzson

MODELICA pelab