**Planning and design for non-expanded-arrays**

**1. Planning**

**1.1. Plan**

***Week 43, OpenModelica Developers Week (Linköping):***

*Meeting 27/10*: Adrian Pop, Per Östlund, Kristian Stavåker, and Alexey Lebedev had a meeting (~1 hour) where the outline of the work and which branch to work on were discussed.

Merge the branch with non-expanded arrays by Alexey:

<https://openmodelica.ida.liu.se/svn/OpenModelica/branches/NonExpandedArrays>

into:

<https://openmodelica.ida.liu.se/svn/OpenModelica/branches/sjoelund-functiontree>

Per and Alexey work together to do the merging.

Kristian Stavåker will look into how to not expand for-equations and how to do the matching without expanding arrays (KS has a meeting with Jens about the matching).

The general idea is to generate a DAE that is has arrays/equations not expanded (needed by us and Equa) and then have a phase to translate it into a DAE that is expanded (because MostForWater needs it).

*Meeting 27/10*: Kristian Stavåker and Jens Frenkel had a meeting (2-3 hours) where the modifications of the matching and sorting algorithms were discussed. Hand compilation of examples on paper.

*Meeting 29/10*: Kristian Stavåker and Jens Frenkel had a meeting (2-3 hours) where it was discussed exactly which functions in the compiler are to be modified. (***More information to be added to document shortly***)

***Week 44:***

Contact between Kristian Stavåker and Jens Frenkel via e-mail and Skype regarding matching and sorting of non-expanded array equations.

***Week 45:***

Kristian is working locally with compiler middle on the sjoelund-functiontree branch. Jens will investigate the code before any commits (+ normal procedure with test suite etc.).

**2. Design principles**

**2.1. Backward compatibility**

The compiler must have arrays non-expanded if and only if it has been launched with +a flag. If this flag has not been set, the compiler must work as before. This leads to the following guidelines:

The cases in matchcontinue which process non-expanded arrays should be guarded by checks of the function RTOpts.splitArays(). This function returns **false** if +a flag has been set and **true** otherwise.

It is preferable to preserve compiler’s data types whenever possible. Changing types makes it harder to guarantee backward compatibility since functions processing such types have to be changed as well. However, sometimes it is unavoidable.

It is also preferable to preserve types of input/output variables of functions, though this is less important since easier to control.

**3. Equation Handling Modifications**

Kristian (18/3-2011):

I have made the following changes in order to handle unexpanded equations as the once in the following model. The changes are on my laptop and not yet committed but I am keeping my local trunk copy updated (on a daily basis).

**model** NonExpandedArray1

**parameter** Integer p=500;

Real x[p];

Real y[p];

Real z[p];

Real q[p];

Real r[p];

**equation**

**der**(x) = 2.3232\*y + 2.3232\*z + 2.3232\*q + 2.3232\*r;

**der**(y) = 2.3232\*x + 2.3232\*z + 2.3232\*q + 2.3232\*r;

**der**(z) = 2.3232\*x + 2.3232\*y + 2.3232\*q + 2.3232\*r;

**der**(q) = 2.3232\*x + 2.3232\*y + 2.3232\*z + 2.3232\*r;

**der**(r) = 2.3232\*x + 2.3232\*y + 2.3232\*z + 2.3232\*q;

**end** NonExpandedArray1;

* In Inst.InstEquationCommonWork and the case /\* equality equations e1 = e2 \*/ the +a flag is retrieved from RTopts.splitArrays and sent as a Boolean flag to Static.elabExp. RTopts.splitArarys is false if no expansion should be done.
* In Inst.condenseArrayEquation a condition, true=RTOpts.splitArrays has been inserted in order to to stop the code from being executed if no expansion should be done.
* In Inst.instArrayEquation some minor changes have been done for not expanding array equations and the code is guarded with the RTOpts.splitArrays flag.
* In the Static module a performVectorization flag has been added to a lot of functions. It was missing in many places.
* In Static.crefVectorize it has been arranged so that no vectorization takes place if the vectorization flag is false.
* In BackendDAE.Equation a new record MULTIDIM\_EQUATION2 has been added. This record simply represents an unexpanded array equation.
* In BackendDAECreat.lower guarded code has been inserted that places the array equations from the array equation array in BackendDAE.DAE into the normal ordered equation array in BackendDAE.DAE. The actual function for performing this transformation can be found in BackendDAEUtil (from BackendDAE.MULTIDIM\_EQUATION to BackendDAE.MULTIDIM\_EQUATION2).
* BackendDAECreate.lowerMultidimeqns and BackendDAECreate.lowerMultidimeqns2 have both been modified.
* In BackendDAEUtil.traversingincidenceRowExpFinder new cases for handling component references and derivative functions call with references to unexpanded arrays have been added. The problem was that in the lowering phase unexpanded arrays are inserted into the environment with variables with the addVar functions but with a subscript list. Then when encountering a variable reference and trying to look it up with getVar we do not have a subscript list so the look up fails. In BackendDAEUtil.traversingincidenceRowExpFinder a hack is made so that we look up the variable with getVar but first we attach a subscript list to the component reference (by using the size in the type information).
* In BackendDAEUtil.incidenceRow a new case has been added for BackendDAE.MULTIDIM\_EQUATION2.

**8. COMPILER FRONT-END: Transformation of array equations and bindings**

If there is an array of a class which has an array binding or array equation, then this modification/equation may transform into something quite different. For example, if we have a class

**class** C

Real x;

Real[dim1] y = fill(x,dim1);

**end** C;

and then we have an array of this class

C[dim2] c;

then in FlatModelica/DAE we have to get something like

Real[dim2] c.x;

Real[dim2,dim1] c.y = {c.x[i] **for** i **in** 1:dim2, j **in** 1:dim1};

Please note that binding

c.y = fill(c.x, dim1);

would not be correct here since fill(c.x, dim1) has dimensions [dim1,dim2] while c.y has dimensions [dim2,dim1].

Other examples of modifications/equations transforming into different form are:

**Scalar-array operations:**

**class** C

Real x;

Real[dim1] y;

Real[dim1] z = x\*y;

**end** C;

C[dim2] c;

would result in

Real[dim2] c.x;

Real[dim2,dim1] c.y;

Real[dim2,dim1] c.z = {c.x[i]\*c.y[i,j] **for** i **in** 1:dim2, j **in** 1:dim1};

**Matrix and vector operations:**

**class** C

Real[dim1,dim1] x,y;

Real[dim1,dim1] z = x\*y;

**end** C;

C[dim2] c;

would result in

Real[dim2,dim1,dim1] c.x;

Real[dim2,dim1,dim1] c.y;

Real[dim2,dim1,dim1] c.z = {(sum c.x[i,j,l]\*c.y[i,l,k] **for** l **in** 1:dim1) **for** i **in** 1:dim2, j **in** 1:dim1, k **in** 1:dim1};

**8.1 Processing of array modifications**

In all the above examples, modifications to array components were given in declarations of those components. Of course, it does not have to be that way; a component can be modified in a declaration of some higher component. For example,

**class** C

Real[dim1] x,y;

**end** C;

**class** A

Real z;

C[dim2] c(y = z\*c.x);

**end** A;

A[dim3] a;

would result in

Real[dim3] a.z;

Real[dim3,dim2,dim1] a.c.x;

Real[dim3,dim2,dim1] a.c.y = {a.z[i]\*a.c.x[i,j,k] **for** i **in** 1:dim3, j **in** 1:dim2, k **in** 1:dim1};

In the expanded case, array bindings are split into bindings for separate array elements. Each separate step of such splitting in examples like above can be performed even if one does not know total number of dimensions of arrays involved. E.g., if we have modification c.y = z\*c.x within some element of the array a, then we need to know neither the total number of dimensions of array a, nor the total number of dimensions of c.x within class A to deduce that c.y[i] = z\*c.x[i] and c.y[i,j] = z\*c.y[i,j] . Then we can add the prefix a[k] to this and get that a.c.y[k,i] = a.z[k]\*a.c.x[k,i] and a.c.y[k,i,j] = a.z[k]\*a.c.x[k,i,j].

In the case of non-expanded arrays, a kind of inverse transformation is needed. It seems that it can be done as follows: If there is a modification of the form

x = F(y,z,…)

in a class declaration, and then there is an array of that class of dimensions [d1, …, dn], then the modification must be transformed as follows for that array:

x = {F(y[i1,…,in],z[i1,…,in],…) **for** i1 **in** 1:d1, …, **for** in **in** 1:dn}

The same can be applied to array equations.

**8.1.1 Technical details of processing of array modifications**

I have tried to apply the procedure described above (replacing x = F(y,z,…) withx = {F(y[i1,…,in],z[i1,…,in],…) **for** i1 **in** 1:d1, …, **for** in **in** 1:dn}) in Inst.instElement, applying the modification to SCode.Mod . There turned out to be some complications:

1. Changing cref to cref[i1,…,in] should be done only for crefs inside the component where the modification occurs. If the cref is outside of that component, than it may happen that indices i1,…,in should not be applied to it, or that not all of them should be applied.

Examples:

Usual situation: applying indices:

**class** C

Real x;

Real[dim1] y = fill(x,dim1);

**end** C;

C[dim2,dim3] c;

should produce something like

Real[dim2,dim3] c.x;

Real[dim2,dim3,dim1] c.y = {fill(c.x[i1,i2],dim1) **for** i1 **in** 1:dim2, i2 **in** 1:dim3};

Cref higher in the hierarchy:

**class** C

Real x;

**end** C;

**class** A

**class** B

C c(x = A.y);

**end** B;

**constant** Real y;

B[3] b;

**end** A;

**class** T

A[2] a(y = {1,2});

**end** T;

should produce something like:

Real[2] a.y = {1,2};

Real[2,3] a.b.c.x = {a.y[i1] **for** i1 **in** 1:2, i2 **in** 1:3}

Here, only the index i1 is added to a.y, while i2 is left out.

(By the way, in the case of expanded arrays, OMC processes this case incorrectly. I have added a bug to the tracker.)

2. If a subcomponent has an **each**-modification, then the indices which should be added are InstDims (again, unless the situation is as in the above case). But the list of for-indices is bigger and may be not yet known when the modification is processed. For example:

**class** C

Real z;

**end** C;

**class** B

C[dim3] c;

**end** B;

**class** A

Real x;

B[dim2] b(c(**each** z = x));

**end** A;

A[dim1] a;

should produce

Real[dim1] a.x;

Real[dim1,dim2,dim3] a.b.c.z = {a.x[i1] **for** i1 **in** 1:dim1, i2 **in** 1:dim2, i3 **in** 1:dim3}

Here, dim3 is not known when modification is found; when a.b.c is processed (and dim3 is known), the compiler (in its current form) has lost the information about which of the indices must be applied to a.x .

These examples demonstrate the need to have some possibility of lookup while applying array transformations to modifications, or maybe, to add some additional information to data types representing modifications (information about how they must be transformed). The transformations should probably be applied to DAE.Mod rather than to SCode.Mod, but I don’t really know when they can be applied.

***OLD TEXT FOLLOWS***

**3. COMPILER FRONT-END: Current OMC implementation**

(This is some information for those who are not very familiar with the way OMC works.)

OMC front-end produces what is essentially flat Modelica in the form of elements of the type DAE.DAElist . Even though DAE stands for “differential algebraic equations”, elements of these lists (of the type DAE.Element) can also represent variables as records DAE.VAR. This record contains variable’s subscripts as a list of DAE.Subscript.

The uniontype DAE.Subscript (which is used not only in DAE.VAR) can be one of three records: INDEX (which is supposed to represent an index), SLICE (which is supposed to represent a range of indices) and WHOLEDIM (which is supposed to represent the whole index range available for a given variable and a given index position, like the first position in the expression a[:,1]).

All the DAE.VAR produced by the front-end as variables within a model (not within a function) represent scalar variables (in particular, scalar elements of arrays) and have subscripts of the form INDEX with index expression being an integer constant.

When instantiating a variable, the uniontype Prefix.ComponentPrefix is used to represent the variable whose subcomponent the variable being instantiated is. For example, in the model

model M

class A

Real b;

end A;

A[2] a;

end M;

the variable a is instantiated with an empty prefix, the variable b is first instantiated with prefix a[1], then with prefix a[2].

Subscripts of Prefix.ComponentPrefix are represented by a list of integers.

**4. COMPILER FRONT-END: Implemented (at least, partially) design changes**

**4.1. Representation of subscripts of non-expanded arrays.** Subscripts of non-expanded array variables are represented by

DAE.WHOLE\_NONEXP (exp= dim),

where dim is the dimension (of type DAE.Exp). This requires change of the type of the 14th input variable in the functions Inst.instVar, instVar\_dispatch, instVar2 and 15th input variable in the function instArray from list<Integer> to list<DAE.Subscript>. Changes in these functions (mostly in instVar2) are also needed of course.

Comments:

Alexey: 1)Originally, SLICE(RANGE(1:dim)) was used instead of WHOLE\_NONEXP (dim). But RANGE was an overkill (I thought that it could be useful for the future development of Modelica – for example, if one day Modelica allows array ranges not to start with 1 or to have step different from 1) and conflicted with range processing. The cases WHOLE\_NONEXP(Exp exp) should probably be combined with WHOLE() into WHOLE(Option<Exp>) in the future.

2) How should enumeration ranges be represented? I have not worked on this yet.

**4.2. Evaluation of dimensions.** The dimension expressions are evaluated if it is possible. They are left unevaluated only if they cannot be evaluated. It means that, for example, the model

model M

parameter Integer p=4;

Real r[p];

end M;

produces variable r with subscript WHOLE\_NONEXP (4), while the model

model M

parameter Integer p;

Real r[p]

end M;

produces variable r with subscript WHOLE\_NONEXP (p).

Comments:

Alexey: It may be preferable to keep leave dimensions unevaluated even when they can be evaluated. For example, it would make it easier to check that the model is balanced for all values of parameters.

**4.3. Prefix subscripts.** The type of subscripts variable in Prefix.ComponetPrefix is changed from list<Integer> to list<DAE.Exp>. Functions in PrefixUtil package are modified accordingly.

**5.1. Subscripts/dimensions type changes.** It will probably be needed to change types of variables representing subscripts or array dimensions from Integers to more complex types in the following places: Values.ARRAY, DAE.ARRAY\_EQUATION, DAE.INITIAL\_ARRAY\_EQUATION.

***END OF OLD TEXT***