**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. 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. COMPILER FRONT-END: Further changes**

**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.

**6. COMPILER MIDDLE: Implementation design**

- We wish to separate for-quations and array-equations. Below we have an *array-equation* followed by a *for-equation*.

(1) dp = **der**(p);

(2) **for** i **in** 2:n-1 **loop**

**der**(dp[i]) = c^2 \* (p[i+1] - 2 \* p[i] + p[i-1])/dL^2;

**end** **for**;

In DAE.Element a new record has been added that represents for-equations of type (2). For handling for-equations with several equations inside, a transformation must be inserted in the front-end that divides a multiple-equation for-equation into several for-equations containing only one equation each. This should always be possible to do since the solving order of equations, for instance inside a for-equation, is not fixed. The new record in DAE.Element:

**record** FOR\_ARRAY\_EQUATION "An initial array equation"

**list**<Dimension> dimension "dimension sizes";

Absyn.ForIterators iterators;

Element eq;

ElementSource source "the origin of the component/equation/algorithm";

**end** FOR\_ARRAY\_EQUATION;

There is already a record, ARRAY\_EQUATION, in DAE.Element for array-equations of type (1).

**record** ARRAY\_EQUATION " an array equation"

**list**<Dimension> dimension "dimension sizes" ;

Exp exp;

Exp array;

ElementSource source "the origin of the component/equation/algorithm";

**end** ARRAY\_EQUATION;

(There is also an INITIAL\_ARRAY\_EQUATION record.)

In BackendDAE.Equation a new record has been added for for-equations. There is already one for array-equations of type (1), ARRAY\_EQUATION.

**record** FOR\_ARRAY\_EQUATION

Integer index "index ; index in arrayequations 0..n-1" ;

Absyn.ForIterators iterators;

**list**< .DAE.Exp> crefOrDerCref "crefOrDerCref ; CREF or der(CREF)" ;

.DAE.ElementSource source "origin of equation";

**end** FOR\_ARRAY\_EQUATION;

* dlow = BackendDAECreate.lower(dae,funcs,**true**,**true**);

Mapping of data structures.

* m = BackendDAEUtil.incidenceMatrix(dlow);

In function DAEUtil.incidenceRow we need to have cases for ARRAY\_EQUATION and FOR\_EQUATION. In function BackendVariable.getVar (called from traverseIncidenceRowExpFinder) we need to be able to handle multi-dimensional variables/arrays.

For creating the incidence matrix several modifications have to be done.

Right now we iterate over the equation list and for each equation we

investigate which variables (state and not state) this equation depend

on. We create a row in the matrix for each equation with 1's for each

variable this equation depends on.

With our modifications we also want to iterate one equation at a time.

But new cases have to be inserted in the function incidenceRow for

non-expanded array-equations and for-equations. As input to this

function we have a list of variables, potentially array variables with

several dimensions. We have to check in the for- or array-equation if at

least one array item is present, if so we should mark in the matrix with

a 1.

We might want to have an incidence matrix with array slices, not

just whole arrays. If we want to match against array slices we need to inspect the equation and see what kind of slices are present. This inspection need to be done before calling incidenceRow since incidenceRow wants a list of variables (potentially array or array slices) as input. So an extra step has to be inserted.

* mT = BackendDAEUtil.transposeMatrix(m);

No changes needed.

* (v1,v2,dlow\_1,m,mT) = BackendDAETransform.matchingAlgorithm(dlow,m,mT, (BackendDAE.INDEX\_REDUCTION(),BackendDAE.EXACT(), BackendDAE.REMOVE\_SIMPLE\_EQN()),funcs);

- checkMatching

We need to check the dimensions of the arrays that are part of the

matching, add this number to the rest of the number of variables. Then compare the number of equations (the dimensions of possible array-equations and for-equations should be counted as well) to the number of variables to be solved.

* (comps) = BackendDAETransform.strongComponents(m,mT,v1,v2);

Hopefully no changes needed.

* simcodegen(cache,env,classname,p,ap,daeimpl,dlow\_1,v1,v2,m,mT, comps);

**7. COMPILER BACK-END (code generator): Implementation design**

indexed\_dlow=BackendDAEUtil.translateDae(dlow,NONE());

indexed\_dlow\_1=BackendDAEUtil.calculateValues(indexed\_dlow);

SimCode.generateModelCode(p,dae,indexed\_dlow\_1,Env.getFunctionTree(cache),classname,cname\_str,file\_dir,ass1,ass2,m,mt,comps,SOME(simSettings));

simCode:=createSimCode(functionTree,outIndexedBackendDAE,equationIndices, variableIndices, incidenceMatrix, incidenceMatrixT, strongComponents, className,filenamePrefix,fileDir,functions,includes,libs,simSettingsOpt); callTargetTemplates(simCode);

**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.