SubSystem in Alpha

In this tutorial, we will present how to write structured alpha programs with subsystems, and we will present the associated transformations.

Syntax of Use Equation (without extension domain)

Let us assume that we want to compute the mean of the values of a vector. It is feasible through the following Alpha system:

```
affine mean {N | N>0}
input
    float A {k | 0<=k<N};
output
    float C {|};
local
    float temp {|};
let
    temp = reduce(+, [k], A[k]);
    C = temp / N;
.
```

However, let us assume that you already have another Alpha system which computes the sum of the elements of a vector. It is possible to use this affine system (instead of rewriting its equation in the main system), by calling it through a **use equation**:

```
affine sum {P| P>0}  // Computes the sum of the elements of a vector of size
P
input
    float vect {i | 0<=i<P };
output
    float Res;
let
    Res = reduce(+, [k], vect[k]);
.

affine mean {N | N>0}
input
    float A {k | 0<=k<N};
output
    float C {|};
local
    float temp {|};
let
    use sum[N] (A) returns (temp);        // Compute "temp" using the system
```

```
"sum"
    C = temp / N;
.
```

The system “mean” is calling the system “sum” (which is called a subsystem). The subsystem is called with the parameter “N” and the input “A”. After doing its computation, the result of “sum” will be stored inside the local variable “temp”.

In general, the syntax of a use equation is the following:

```
use subsystem_name[list of parameters] (list of input expressions) returns
(list of output variables);
```

If your subsystem has several parameters/inputs/outputs, you have to provide them in the order in which they are declared.

Extension domain

Let us assume that you have a system which computes a dot product of two vectors:

```
affine dotProduct {N | N>0}
input
    float v1 {k | 0<=k<N};
    float v2 {k | 0<=k<N};
output
    float Res {|};
let
    Res = reduce(+, [k], v1[k]*v2[k]);
.
```

If you want to compute a matrix vector multiplication using this affine system, you will need to instanciate it once per rows of the matrix. Thus, you will need a parametrised number of call to the “dotProduct” system.

It is possible to do it by using an extension domain:

```
affine dotProduct {N | N>0}
input
    float v1 {k | 0<=k<N};
    float v2 {k | 0<=k<N};
output
    float Res {|};
let
    Res = reduce(+, [k], v1[k]*v2[k]);
.
```

```
affine matrixVectorProduct {R,S | (R,S)>0}
input
    float mat {i,j | 0<=i<R && 0<=j<S };
    float vect {j | 0<=j<S};
output
    float vectRes {i | 0<=i<R};
let
    use {k | 0<=k<R} dotProduct[R] ( mat, (k,j->j)@vect) returns (vectRes);
.
```

The set “{k | 0≤k<R}” before the subsystem name is called the **extension domain**. We are calling the system “dotProduct” once, for each instance of “k” in the extension domain. We can use the indexes of the extension domain to parametrize the parameters, inputs given to the subsystem and the outputs computed by the subsystem:

1. the indexes can be used to specify the parameters (ex: “R+k”)
2. the first dimensions of the input expressions correspond to the dimensions of the extension domain. For example, each row of “mat” will be sent to a different instance of the subsystem (ex: in the previous example, the third instance of “dotProduct” will receive “(j→3,j)@mat” and “(j→j)@vect” as inputs).
3. the first dimensions of the output variables correspond to the dimensions of the extension domain. All the results from every subsystem call are gathered inside common variables (ex: “vectRes[3]” is the output of the third instance of “dotProduct”)

Apart from the compatibility of dimensions, the input expressions must be defined at least on the points asked by the subsystem, and the output variable must be defined on a subset of the domain of the subsystem output.

Transformations involving subsystems

InlineSubSystem: Inline the equations of a subsystem inside the affine system calling it. The use equation of the main system is replaced by the equations of the subsystem (which are adapted), and new local variables are added.

The command is: `void InlineSubSystem(Program program, String systemName, String label)` where `label` is the label of the inlined use equation.

OutlineSubSystem: Given a list of equations of an affine systemm, outline them inside a new system and replace these equation by a use equation. The current version (July 2014) do not allow to specify an extension domain, however this is a work in progress.

The command is `void OutlineSubSystem(Program program, String system, String listEquations)` where `listEquations` is the list of label of the equations to be outlined.

From:
https://www.cs.colostate.edu/AlphaZ/wiki/ - **AlphaZ**

Permanent link:
https://www.cs.colostate.edu/AlphaZ/wiki/doku.php?id=tutorial_subsystem&rev=1405360767

Last update: **2014/07/14 11:59**