Theoretical basis of the Pipeline and PipeControl package

Tanguy Risset Patrice Quinton*

Version of December 1, 2009

Introduction

This document is intended to provide the precise semantics of the functions of the `Pipeline.m` package, together with their implementation details and examples of use. These functions are: `pipeline, pipeall, pipeInput, pipeOutput, pipeIO`.

1 ToAlpha0v2

1.1 Space time separation

See `ToAlpha0v2.m`.

Call to `spaceTimeDecomposition`, with time position and space position. An option (exceptions) allows some variables to be ignored.

Compute the list of variables to consider : all variables, except output of use expressions. Then, call function `spaceTimeCase` for each variable.

This function check that the variable is defined, and that the definition is a case expression. If it is not the case, it returns the system unchanged.

Otherwise, it calls `makeSTCase`. This function will return a new expression that will replace the old one.

Two cases to be considered: one or more than one branches.

If the expression has only one branch. We compute the domain of the RHS expression (using `expDomain`). We compute the intersection of this domain and of the LHS domain (actually, the declaration domain of the variable), and we signal if the expression domain is not included in the LHS domain. Then, we return this intersection domain.

If there are more than one branch, we proceed recursively. Let d_0 the domain of the first branch. Assume that the $n-1$ remaining branches are in ST form, and let $d_i : exp_i$ be the other branches.

*Last modification: 22 nov. 2009

compute the intersection d_p of the space projection of d_i and the space projection of d_i ;if d_p is empty, we add $d_i : exp_i$ to the list of restrictions, and we keep d_0 unchanged for the next round. if not, we compute $d = d_0 \cap d_i$; and $dinter = d_i\ d$

2 PipeControl

The `pipeControl` function aims to optimize a control definition equation. Such an equation has the form

- ```
  Vctrl[ t, p1, ... ] =
  case
    domProcs:
  case
     domTime1: exp1;
     domTime2; exp2;
  esac;
  esac;
  ```

(It is called by the `pipeAllControl` function, which finds out all variables that are supposed to be control equations. So, if called directly, the property is not necessarily checked, and the equation may not fit this format.)

The idea is as follows. First, we try to find out a separating hyperplane for both expressions. The idea would be to find out either a processor domain, so that the value of the control signal is related to the processor position, or a time plane, so that the value of the control could be defined by a controller and broadcasted to all processors.

Once such an hyperplane is found, it gives a direction along which to propagate the `True` and `False` values to the boundaries of the domain.

We first compute the direction vector of the hyperplane, and then, we compute its null space vectors. We select these vectors in such a way that their first component (related to time) is not zero, otherwise, that would mean that we cannot pipeline, but only delocalize the control.

2.1 Checking

The function checks the following conditions:

1. The variable to pipeline exists (message `pipeControl::unknownvar`).
2. The dimension of the variable is less than or equal to 4.
3. The first dimension is the time.
4.
  ```

3 Pipeline

3.1 Explanation of the transformation

The pipeline transformation is used to transform a program with a definition of a variable var_1 which contains an expression $expr$ implying a non uniform dependence f ($Ker(f) \neq \emptyset$): $var_1[z] = F(expr[f(z)])$, into an equivalent[1] program which contains a uniform dependence d[2] for the definition of the same variable var_1.

The principle of the transformation is the following, the user gives :

- the name of a new variable to be created (pipeline variable: $pipeExpr$);
- the exact instance of the expression to be pipeline ($expr[f(z)]$ in var_1);
- and a pipeline vector d.

In the transformed program, the variable $pipeExpr$ is defined on the whole domain D where the expression $expr[f(z)]$ is used. $pipeExpr[x]$ is initialized to $expr[f(x)]$ on the *border*[3] *of D along d* (we call this border B). This value is duplicated (we say *pipelined*) along d on the rest of the domain D ($pipeExpr[z] = pipeExpr[z-d]$). Then, the corresponding part of the definition of var_1 is replaced by: $var_1[z] = pipeExpr[z]$ (see figure 1).

This transformation cannot be done for every pipeline vector. Note that the transformation imply that the values $pipeExpr[z + kd]\ \forall z \in B,\ k \in N$ will correspond to the instance $expr[f(z)]$, thus we must have $expr[f(z)] = expr[f(z + kz)]\ \forall z \in B,\ k \in N$. This is verified if $d \in Ker(f)$. This imply that f must be non invertible. Note that the pipeline vector may be a not primitive vector, but this may induce warning during analysis of the resulting program because it correspond to the assumption that the domain of use of the expression is not flat.

3.2 Implementation

The `pipeline` function should be used by programmers only, the users should use `pipeall`. In the parameter of the function, the name of the variable and the pipeline vector are catched in the same expression :`"Name.function"` where the function is the translation in the direction of the pipeline vector.

At the moment, the domain of the new variable build is the intersection of the context domain of the expression to pipeline (`getContextDomain`) with the domain of the expression to pipeline (`expDomain`). If the pipeline vector is not in the kernel of the dependence function, the transformation is aborted.

[1] equivalence to be proved

[2] or several uniform dependencies

[3] at the points where we cannot retrieve one more pipeline vector without getting out of D

$expr[f(z)]$ $expr[f(z)]$

B

$Var_1(z)$ $pipeVar(z)$

$Var_1(z)$

D

Figure 1: Example of pipeline Transformation

3.3 pipeall

`pipeall` is like a pipeline more convivial. If an expression is present in several places in the definition of a variable, or even in several definition of different variables, `pipeall` will pipeline all occurrences of this expression. In that case, the domain on which is defined the variable is the union of all the domains which would result of each individual pipelines. The main difference with pipeline are the way the arguments are asked (see `?pipeall`). Be careful, you cannot expressed the expression to be pipelined using the array notation, you must specify `expr.(x->f(x))` instead of `expr[f(x)]`.

4 Pipeline of Inputs and Outputs

the `pipeline` transformation pipelines a dependance which was originally a broadcast (dependence non invertible). We may need to pipeline an expression even if it is not broadcasted, this is particularly useful when the expression to pipeline must be input (resp. output) in an architecture, in which they must go through some cells before being used in a computation (array with a loading phase). This can be done with the `pipeIO` function, which should be used through the functions `pipeInput` and `pipeOutput`. `pipeIO` perform a *routing*. It takes some data at some place of the iteration space and bring it into another place of the iteration space.

4.1 PipeInput

The user gives:

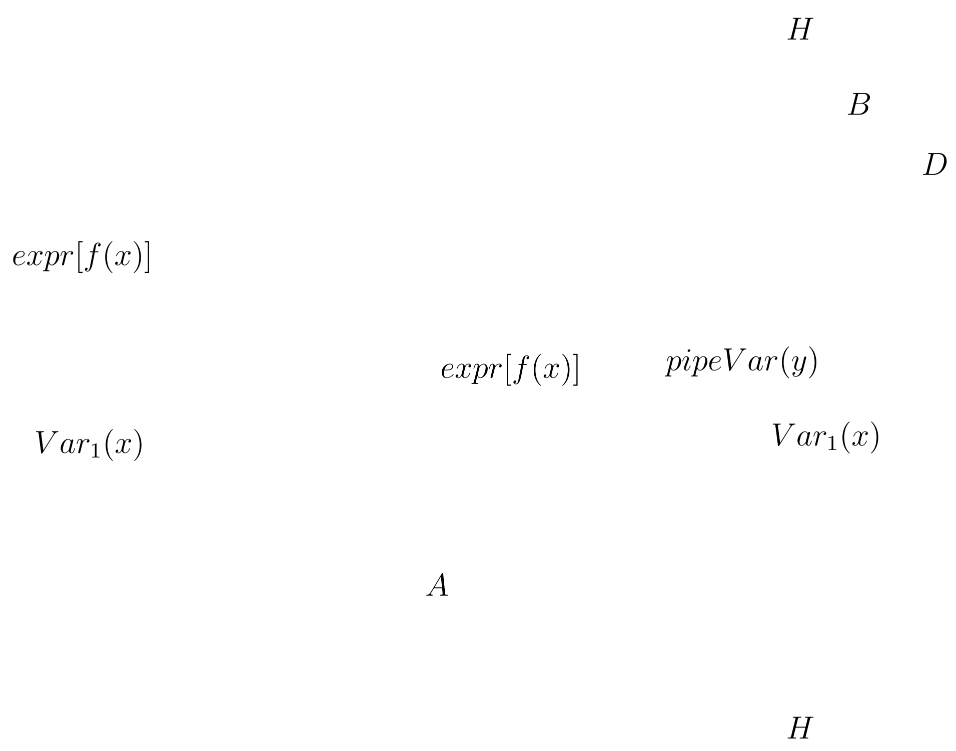

Figure 2: Example of input pipe with `pipeInput`

- the name of a new variable to be created (pipeline variable: *pipeExpr*);
- the exact instance of the expression to be pipeline ($expr[f(z)]$ in $var_1[f(z)]$, note that f may be non-singular);
- a pipeline vector d;
- and the half space in which the pipeline is to be performed (bounded by an hyperplane H).

Pipelining an expression as an input consists in the following transformation: we suppose the expression $expr[f(x)]$ is used an expression somewhere x ($Var[x] = expr[f(x)]$ on domain A, see figure 2). After the transformation, the expression *expr* is used somewhere else y (along the bounding hyperplan H) and the value is propagated to location x by the pipeline variable ($Var[x] = pipeExpr[x]$ on the original domain A, $pipeExpr[z] = pipeExpr[z-d]$ inside the pipeline Domain D, and $pipeExpr[y] = epxr[f[x]]$ on the border B). This transformation is very close to the usual pipeline transformation

This transformation is illustrated on figure 3 and 4. figure 3 represents the original program and figure 4 represents the program after the execution of the command: `pipeInput["C", "b.(i,j->i)","B1.(i,j->i+1,j+1)", " {i,j | i >= 0 } "]` and normalization. The example also corresponds to the illustration of figure 2. $Var1$ is `C`, $expr[f(x)]$ is `b.(i,j->i)`, d is `(1,1)` (represented by `(i,j->i+1,j+1)`), $pipeVar$ is `B1`, and H is `{i,j | i >= 0}`.

```
system silly: {N | N>1}
                (a : {i,j|1 <= i,j <= N} of boolean;
                 b : {i|1 <= i <= N} of boolean)
     returns (c : {i|1 <= i <= N} of boolean);
  var
        C :  {i,j|1 <= i <= N; 0<= j <=N} of boolean;
  let
C[i,j] = case
       {|j=0} : b[i];
       {| j>=1} : C[i,j-1] + a[i,j];
         esac;
      c[i]=C[i,N];
  tel;
```

Figure 3: simple program before the use of pipeInput

4.2 PipeOutput

The user gives (as for `pipeInput`:

- the name of a new variable to be created (pipeline variable: $pipeExpr$);
- the exact instance of the expression to be pipeline ($expr[f(z)]$ in $var_1[f(z)]$, note that f may be non-singular);
- a pipeline vector d;
- and the half space in which the pipeline is to be performed (bounded by an hyperplane H).

Pipelining an expression as an output consists in the following transformation: we suppose we use at some place x an expression which was produce at some place y ($Var[x] = expr[y]$ on domain A). In the transformed program, we use this expression in another variable at place y ($VarPipe[y] = expr[y]$) and the value in pipelined in $VarPipe$ until another place $f(x)$ where it is consumed by Var ($Var[x] = VarPipe[f(x)]$ on domain B, see figure 5 for example of output pipe).

This transformation is illustrated on figure 6. figure 3 represents the original program and figure 4 represents the program after the execution of

```
......
var
  B1 : {i,j | (j+1,0)<=i<=j+N; j<=0; 2<=N} of integer;
  C : {i,j | 1<=i<=N; 0<=j<=N} of boolean;
let
  B1[i,j] =
      case
        {| i=0; -N<=j<=-1; 2<=N} : b[i-j];
        {| 1<=i<=j+N; j<=0; 2<=N} : B1[i-1,j-1];
      esac;
  C[i,j] =
      case
        {| 1<=i<=N; j=0; 2<=N} : B1;
        {| 1<=j} : C[i,j-1] + a[i,j];
      esac;
.......
```

Figure 4: Program of figure 3, after use of `pipeInput`: `pipeInput["C", "b.(i,j->i)","B1.(i,j->i+1,j+1)", " {i,j | i >= 0 } "]`

the command: `pipeOutput["c", "C","C1.(i,j->i+1,j+1)", "{i,j| i <= N}"]`. The transformation performed is illustrated on figure 5 where Var is `c`, $expr[f(x)]$ is `C.(i,j->i,j)`, d is `(1,1)` (represented by `(i,j->i+1,j+1)`), $varPipe$ is `C1` and H is . `{i,j| i <= N}`.

4.3 Implementation

Both `pipeOutput` and `pipeInput` are implemented by the same function : `pipeIO`, we will briefly described the implementation of this function here. The fact that the pipeline is an input pipe or an output pipe is determined by the scalar dot of the pipeline vector and the normal to the hyperplane H bounding the half space. If the pipeline vector goes *towards* H then, this must be an output pipe, if it *comes from* H, this must be an input Pipe

Three domains are distinguished:

- the domain of the original expression $expr[f(z)]$ (that we will call $domExpr$), which is the domain where the pipeline is initialized in the case of an output pipe and the domain where the pipeline ends in the case of an input pipe (domain A on figure 2 and 5).
- the pipeline domain (that we will call $realDomPipe$) which is the domain on which the value is pipelined (domain D on figure 2 and 5).
- the domains where the pipeline ends (that we will call $pipeEndDom$), which is in fact the domain where the pipeline is initialized in the case

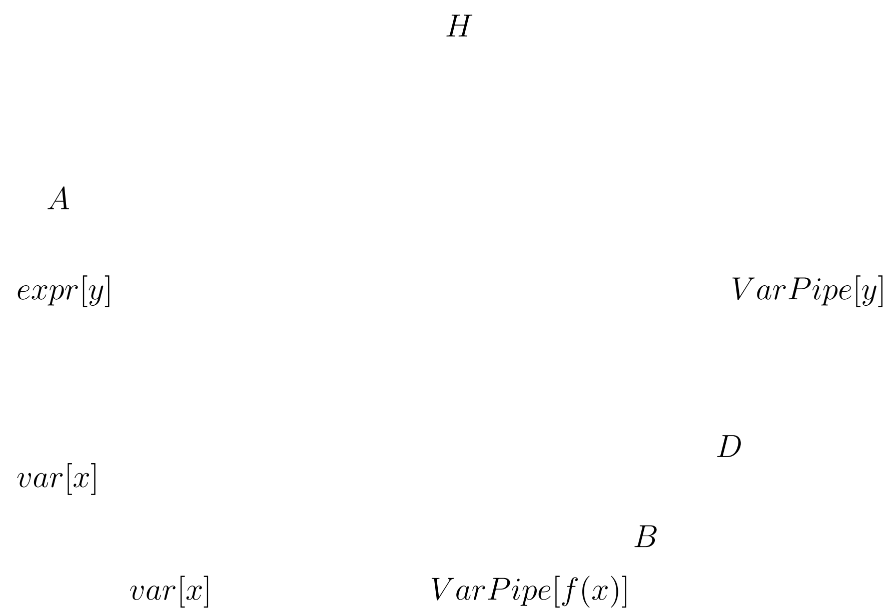

Figure 5: Example of output pipe with `pipeOutput`

```
var

  C1 : {i,j | j-N+1<=i<=N; N<=j; 2<=N} of integer;
.........
  C1[i,j] =
       case
         {| 1<=i<=N; j=N} : C;
          {| j-N+1<=i<=N; N+1<=j} :  C1[i-1,j-1];
       esac;
  c[i] = C1[N,-i+2N];
tel;
```

Figure 6: program of figure 3, after use of pipeOuput: `pipeIO["c", "C","C1.(i,j->i+1,j+1)","{i,j| i <= N}"]`

of an input pipe (domain B on figure 2 and 5).

domExpr is computed as the intersection of the context domain of the expression to pipeline (`getContextDomain`) and the domain of the expression itself (`expDomain`). *realDomPipe* is computed by adding to *domExpr* a ray (which is the pipeline vector in the case of an output pipe and its opposite in the case of an input pipe) and then by intersection the resulting domain with the half space bounded by H. *pipeEndDom* is computed by shifting *realPipeDomain* by the pipeline vector in case of input pipe (resp. its opposite in case of output pipe) and retrieving the *realDomPipe*.

there remains the problem of founding, given on index point in *pipeEndDom*, what is the corresponding point in *domExpr* (resp. the other way around in the case of input pipe). This is done by observing the following fact. If we find a function of the indices whose value is constant on the pipeline path and which is perfectly determined by the pipeline path (unique for each pipeline path), then giving this value will determine a unique antecedent point z_1 in *exprDom* and a unique antecedent point z_2 in *pipeEndDom*. Hence, we will be able to use the `inverseInContext` function is order to find for a point in *pipeEndDom* the corresponding point in *exprDom*. the function to build must have a square matrix thus we take a square $n \times n$ matrix of rank $n-1$ for which the kernel is generated by the pipeline vector. And it works... The use of `inverseInContext` impose that the domains of the expressions to pipeline must be flat (dimension $n-1$).