New User Interface for Petit and Other Extensions

User Guide

Wayne Kelly, Vadim Maslov, William Pugh,
Evan Rosser, Tatiana Shpeisman and David Wonnacott

omega@cs.umd.edu
http://www.cs.umd.edu/projects/omega

December 5, 1996

Contents

1 Introduction

This document describes version 1.00of petit , a version of Michael Wolfe's Tiny Tool as extended by the Omega Project at the University of Maryland. Petit is a research tool for doing dependence analysis and program transformations on programs in the petit language, a simple array language that lacks features that complicate dependence analysis. Petit no longer contains the loop restructuring features of the original Tiny.

Petit and other Omega Project software can be retrieved via FTP from `ftp.cs.umd.edu` in the `/pub/omega` directory. More information on the Omega Project can be found on the World Wide Web at `http://www.cs.umd.edu/projects/omega`.

Features of version 1.00include:

- Nonlinear dependence analysis based on uninterpreted function symbols
- Scalar analyses to enable more accurate analysis (induction variable recognition, forward substitution)
- Scalar/array expansion and privatization
- A Motif-based X Window System interface that allows browsing of dependences and dependence relations
- An interface to facilitate the generation of constraints for program analysis
- An interface to integrate our dependence analysis into other compilers

Earlier improvements to petit included:

- Value-based dependence analysis
- A Unified system for performing transformations
- Induction variable recognition (IVR) using the Static Single Assignment (SSA) graph
- Forward substitution of scalars
- Elimination of dead assignments to scalars
- Fortran-77 to Petit converter (`f2p`)
- Built-in functions
- Storage classes for variables
- Entry and Exit nodes for dependences involving global variables
- Update (assignment) statements `+=`, `*=`, `max=`, `min=`
- Reduction dependences
- DOANY loop
- Scalar and array expansion to kill anti and output dependences
- Array privatization

2 Compiling petit

Petit is written in C++, and has been compiled with GNU G++ versions 2.5.8 and 2.6.3. Version 2.6.3 is recommended, since its handling of templates produces smaller object files and executables. It may be possible to compile Petit with other commercial C++ compilers; petit does not depend on any features of the GNU `libg++`.

Petit requires bison and flex rather than lex and yacc. The output of bison and flex is included, so you should be able to compile without them unless you make changes to the petit grammar.

Petit requires X11 and Motif to compile the interactive version. A batch-only petit can be built. See the `Makefile` for details. A "batch-only" petit can be built even if you do not have the Motif include files and libraries available. Use the flag `-DBATCH_ONLY_PETIT=1` in the `PETIT_FLAGS` makefile variable. A petit built in this way will only work with the `-R` option (see Section 4).

Petit requires version 1.00 of the Omega Library to work. Both are available for anonymous ftp at `ftp://ftp.cs.umd.edu/pub/omega`.

Petit is distributed with a Makefile that is known to work with Make SunOS 4.1.3 and with GNU Make on Ultrix 4.3. It should compile on most UNIX systems. To compile, go to the `petit/obj` directory. You may want to look at the makefile to adjust paths to X11 include files and the Omega Library and to set the compiler to use. Then, do a `make depend` and then `make petit`.

3 Using petit interactively

To use petit, run `petit` or `petit` *filename*. A window will appear displaying the petit-language program.

3.1 Using The Mouse

When displaying a program, Petit highlights the "current" node in the program being browsed.

Clicking the left button on a node causes it to become the current node being browsed. If the user is not already in the browse menu then they are automatically placed in the browse menu, and when they exit this menu they are returned to the menu they were working in when they clicked the button. If the user is already in the browse dependence menu when they click the left button and the node they click on is involved in a dependence then the first dependence that node is involved in becomes the current dependence being browsed.

The middle button is used to change the dependence currently being browsed. If the user is not already in the browse dependence menu then they are automatically placed in the browse dependence menu, and when they exit this menu, they are returned to the menu they were working in when they clicked the button. The middle button can only be used to select nodes that could possibly be involved in dependences. The first dependence involving the selected node, if any, becomes the current dependence being browsed. The middle button also has the affect of pulling up the pop up dependence window described below.

The right button is used to toggle whether this dependence window is displayed.

3.2 Dependence status

In any listing of a dependence, the status of a dependence may be indicated by a set of letters in square brackets at the right end of the line describing the dependence. These status flags arise from two types of extended analysis. First, we attempt to determine which dependences are based on values and which are based solely on sharing memory locations. Dependences based solely on shared memory locations are termed *dead*. We also check to see if there exist interesting assertions that can be added to the program so as to eliminate the dependence. The meanings of the letters are as follows:

`M` - The dependence is *memory-based*; that is, the source and sink may access the same location in memory.

`V` - The dependence is *value-based*. There is a flow of values from the source to the sink. In other words, the source and sink access the same memory location and there is no intervening write to the same location.

`i,I,j,J` - The dependence is *inexact*. The value-based dependence equations could not be solved exactly due to nonlinear terms in the program. The form of the dependence is:

- `i` one or more exact conjuncts, and one or more conjuncts including *unknown*.
- `I` one or more conjuncts including *unknown*.
- `j` one or more exact conjuncts and one conjunct that is simply *unknown*.
- `J` the relation is simply *unknown*.

`z,Z,?` - The dependence may be *zappable*; that is, there may exist conditions under which the dependence does not exist despite the fact that the loop carrying the dependence executes more than one iteration.

- `Z` Conditional analysis yields exact conditions that zap the dependence. Adding an assertion that contradicts these conditions will eliminate the dependence.
- `z` Conditional analysis produces conditions that zap the dependence but contain some *unknowns*. It is still possible to zap the dependence with the given conditions, but there may be other zapping conditions as well.
- `?` The result the conditional analysis produces is simply *unknown*. While it may be possible to eliminate this dependence, the system is unable to describe conditions that do so.

`O` - The dependence is involved in a cycle of value-based flow dependences.

`o` - The dependence is involved in a cycle.

3.3 The Dependence Window

The dependence window is a pop up window that appears on top of the main window. It contains a list of the dependences involving the node currently being browsed. By default, dependences involving the current browse node as either the source or destination node are shown in the window. The dependences involving the current browse node as the destination node are separated from the dependences involving the current browse node as the source node are separated by a line of ~'s. Initially the dependences involving the current browse node as the source node are displayed. The dependences involving the current browse node as the destination node can be viewed by scrolling the dependence window up.

Clicking on a dependence in the dependence window will highlight the the source and destination in the program text (if the window is large enough to show both of them.)

The bottom line of the dependence window displays the dependence relation for the current dependence. A dependence relation is available only when using the Omega Test or the Omega-Epsilon test as the dependence analysis algorithm.

3.4 Dependence Filters

The dependence window contains a set of filter options. Clicking on these options toggles whether the corresponding filter should be applied to the dependences shown to the user. If a filter is "on", then dependences of the type corresponding to this filter are shown to the user. Initially all filters are set to "on". The filter settings remain in affect until they are changed, even after the dependence window is closed. The filters affect all dependences that are shown to the user, not just dependences shown in the dependence window, i.e. they affect the existing "cycle" and "next" options for example.

There are four filters which determine which types of dependences are not shown to the user: "flow", "anti", "output" and "reduce" (the last one for reduction dependences.) The four filters "memory", "value", "~carry" and "~cyclic" detemine whether dependencies that are memory-based, value-based, loop independent and not involved in the dependence cycle respectively should be not shown to the user. Finally, the scalar filter determines whether dependences between scalars are shown to the user.

4 Using petit in batch mode

Petit can be used in "batch mode" to produce files of dependences without using the X Window interface. The `-R`*output-file* writes the data dependences for the file being analyzed in *output-file*.

5 Integrating petit with other compilers

If you are interested in incorporating our dependence analysis routines into your software, see the file `include/lang-interf.generic` for instructions.

6 Building relations using Zima interface

The Zima interface facilitates the process of building the relations used for the program analysis. The interface is based on the notation suggested by Hans Zima [Hans Zima and Barbara Chapman *Supercompilers for Parallel and Vector Computers.*ACM Press, 1991]). For example, the memory-based dependence relation from A to B can be written as follows using this notation:

$$[i \to j] : i \in [A] \wedge j \in [B] \wedge A(i) \ll B(j) \wedge A(i) \stackrel{sub}{=} B(j)$$

In the Zima interface an access to the variable in the program at the particular loop iteration ($A(i)$,$B(j)$) is represented as an element of a class AccessIteration. It can be created using one of the two following constructors:

`AccessIteration(a_access a, Relation * r, const Variable_ID_Tuple * t)`

or

`AccessIteration(a_access a, Relation * r, Argument_Tuple _of)`, where

`a_access a` is an access to the memory,
`Relation * r` is a pointer to the relation being built,
`Variable_ID_Tuple *t` is a pointer to the tuple describing a loop iteration at which an access occurs,
`Argument_Tuple _of` is type of the tuple: `Input_Tuple` or `Output_Tuple`.

The following functions allow us to add constraints to the relation:

`access_same_memory(F_And *N, const AccessIteration &AI, const AccessIteration &BJ)` - add the constraints requiring that the memory location accessed in `AI` and `BJ` is the same. ($A(i) \stackrel{sub}{=} B(j)$).

`access_in_bounds(F_And * N, const AccessIteration &AI)` - add the constraints requiring that the loop indices for `AI` are within the loop bounds ($i \in A$).

`precedes(F_And *N, const AccessIteration &AI, const AccessIteration &BJ)` - add the constraints requiring that access `AI` happens before access `BJ` ($A(i) \ll B(j)$).

`precedes(F_And *N, const AccessIteration &AI, const AccessIteration &BJ, int l)` - add the constraints requiring that the access `AI` precedes the access `BJ` at level `l`.

`subscripts_equal_to_tuple (F_And *N, const AccessIteration &AI, const Variable_ID_Tuple *t)` - add the constraints requiring that the subscripts of the array access `AI` are equal to the corresponding elements of the tuple `*t`. The size of `*t` should be equal to the dimensionality of the array.

In all the above functions the constraints are added below the formula node `N`.

The following code demonstrates building a dependence relation shown above using the Zima interface.

```
int input_tuple_size = access_depth(A);
int output_tuple_size = access_depth(B);
Relation dependence(input_tuple_size, output_tuple_size);
AccessIteration AI(A, &dependence, Input_Tuple);
AccessIteration BJ(B, &dependence, Output_Tuple);
```

```
F_And *f = dependence.add_and();
access_in_bounds(f, AI);
access_in_bounds(f, BJ);
precedes(f,AI,BJ);
access_same_memory(f, AI, BJ);
dependence.finalize();
```

7 Command line options

Petit is invoked with any number of command line options, and, (optionally,) a initial file to read in. Options may precede file name or follow it. When debugging is asked for, it is written to the file `trace.out`. The command line options are:

- `-4` Skip "Omega-4" (value-based) analysis.
- `-5` Skip "Omega-5" (nonlinear) analysis.
- `-A` Do some expensive assertions for debugging SSA.
- `-C` Print SCC components.
- `-D` Do dependence analysis debugging: -D[0-4]
- `-F[vmcfoa]` Filter. Print only dependences of types specified:
  - `v` Include value based dependences.
  - `m` Include memory based dependences.
  - `c` Include loop carried dependences.
  - `f` Include flow dependences.
  - `o` Include output dependences.
  - `a` Include anti dependences.
- `-H` Print phi-functions and variable instances.
- `-I` SSA and IVR debug information.
- `-K` Calculate number of dependences.
- `-L` Print affine expressions in [ ].
- `-M[0-3]` Turn on debugging of all Omega Library functions except the omega core.
- `-N` Remove IVR-generated no-op assignment stmts.
- `-O[0-3]` Omega core debugging.
- `-P` Omega Library debug: -P[achgpr][0..4]
  - `a` Include all debugging info.
  - `c` Omega core debugging.
  - `h` Debugging of convex hull calculations.
  - `g` Debugging of Omega Library code generation.
  - `r` Debugging of relation operators.
  - `p` Debugging of basic presburger arithmetic operations.
  - `t` Debugging of transitive closure calculations.
- `-Rfilename` Print dependences to `filename`.
- `-U` Unclassified variables have become IN variables.

- `-Y` Notify about index array occurences.
- `-a`*d* Use data dependence analysis algorithm number *d*. By default the Omega test is used. Currently the following algorithms may be used (identified by algorithm number):
  1 The Omega test.
  2 The Omega test, using the Epsilon test as a preprocessing step.
  3 The Epsilon test, a series of standard dependence tests.
  4 Banerjee's inequalities.
- `-b` Print dependence relations.
- `-d` Make arrays INOUT by default.
- `-f` Do fast and approximate dependence analysis.
- `-g` Ignore Entry/Exit node dependences.
- `-h` Eliminate shadows of shadow expressions.
- `-i` Do induction variable recognition.
- `-j` Only print dependencies due to the arrays of type real
- `-l` Replace affine expr. with norm. form.
- `-m` Set the maximum number of conjuncts in the relation to consider: -m[n]
- `-o` Don't print dead dependences and their flags.
- `-p` Do privatization.
- `-q` Be quiet.
- `-r` Recognize update operations.
- `-s` Skip analysis for scalar.
- `-t` Skip analysis for top level dependences.
- `-u` Panic when there is a non-affine term during value-based or zapping analysis.
- `-w` Set zappable intersting distance: -w[n]
- `-x` Do array expansion.
- `-y` Repeat array expansion (if it's on).
- `-z` Substitute real scalars too.

8 New user interface

8.1 Zapping Dependences

In order to zap dependences in Petit, select the zap option which is under the DD option, which in turn is under the browse option in the main menu. When this is done, Petit displays the conditions that must hold in order for a dependence to exist. The user can then indicate that one of these conditions is known to be false, and Petit will add an assertion negating that condition, and remove the dependence.

Petit can not handle those cases in which the conjunction of the conditions is known to be false, but no individual condition is false. Thus if the conditions displayed are

1. 3 <= n
2. n <= 10

and we know that $n < 3$ holds then we just choose 1. But Petit can not handle the case in which we know that either $n < 3$ or $n > 10$ is true, but neither condition by itself is always true.

9 Changes to Petit language

These are changes made to the original Tiny language. In addition to changes listed below some "syntactic sugar" changes were made. These changes are listed in the Petit grammar file `src/petity.y`.

9.1 Assert Statements

Assert statements can now be added to a Petit program. These can be placed anywhere in the program text and must be true for the entire program. The syntax for these statements is

`assert`(<arithmetic expression> <comparison operator> <arithmetic expression>)

where a <comparison operator> can be any one of $>, <, >=$ and $<=$, and the arithmetic expressions must be comprised of numeric and symbolic constants and arithmetic operators.

9.2 Change of Operator precedences

The exponentiation operator ($**$) now has the highest precedence and the $<, <=, >, >=$ and $<>$ have been assigned the lowest precedence. Thus the expression

$$a ** 2 + b * c < 10$$

is now equivalent to

$$((a ** 2) + (b * c)) < 10.$$

9.3 Built-in functions

Built-in functions can now be used in Petit program. First, built-in function should be declared like

```
builtin real abs(), sin(), myfun()
```

This declaration informs Petit that functions `abs`, `sin`, and `myfun` have no side effect and can be treated as conventional arithmetic operations.

F2t converter automatically declares all standard Fortran-77 functions as `builtin`.

9.4 Storage classes

Now each Petit variable has a storage class. It's needed to tell whether variable is defined on entry to the procedure and whether variable value is used on exit from the procedure. This information is used by the induction variable recognition (IVR) transformations and by the dependence analyzer.

There are following storage classes defined:

- `auto` Automatic variable. No value is assigned to variable on entry to procedure, and value of variable on exit from the procedure is not used.
- `common`
- `static`
- `formal` Common, static or formal (dummy parameter) variable. Variable has a value on entry to procedure and its value on exit from procedure is used. The class `common` corresponds to Fortran COMMON. The class `static` corresponds to Fortran SAVEd. The class `formal` corresponds to Fortran subroutine dummy parameter.
- `in` Variable has a value on entry to the procedure, but its value on exit from the procedure is not used.
- `out` Variable has no value on entry, but its exit value is used.
- `inout` Equivalent to COMMON — both entry and exit values are present.
- `private` Private variable. There exists separate instance of the private variable for each point of iteration space created by loops surrounding the declaration of this variable. Therefore no dependence can be carried by surrounding loops for private variable.
- `builtin` Built-in function — see above.
- `const` Named constant.

When variable is declared its class should precede variable type. If class is omitted then variable is considered to be of the class `auto` for scalars and of class `inout` for arrays. Also if there is a use of scalar variable before its definition in Petit program and no storage class is explicitly assigned to variable, the variable gets `in` class.

F2t converter generates relevant storage classes for Fortran variables.

9.5 Update statements `+=`, `*=`, `max=`, `min=`

Now it's possible to write a la C

```
s += a(i,j) + b(k)
```

instead of

```
s = a(i,j) + b(k) + s
```

We introduced update statements only for both commutative and associative operations (`+, *, min, max`). It was done so because we introduced reduction dependences which may exist only between two updates (see below).

There's no need to rewrite existing Petit programs to get advantage of reduction dependences which are less restrictive than other types of dependences. Petit now can automatically convert assignment statements to semantically equivalent update statements if it is started with `-r` command line option.

For example, assignment `s = s + a(i,j) + 2` or even `s = a(i,j) + s + 2` is converted by Petit to `s += a(i,j) + 2`.

9.6 DOANY loop

This type of loop was suggested by Michael Wolfe in the paper "Doany: Not Just Another Parallel Loop" published at Conference record of 5th Workshop on Languages and Compilers for Parallel Computers, Yale University, August 1992. It instructs computer to execute iterations of the loop in any sequential order but not simultaneously.

When performing parallelization of a loop (either by running menu command Browse.Restr.Par or while doing parallelization of all loops by running System.Auto.AutoParallel) Petit makes loop to be DOANY if all dependences carried by it are reduction dependences.

The DOANY loop is intimately related to *reduction* dependences (see below).

9.7 RETURN statement

A RETURN statement is now allowed to appear at the end of the Petit program. It may contain an expression which is ignored.

`f2p` assigns storage class OUT to Fortran function return variable, so assignments to it won't be eliminated by IVR dead statement eliminator.

9.8 Run-time dimensions and undeclared variables

Array dimensions can be run-time.

Undeclared variables are allowed to appear in the bounds of array being declared. They will be automatically declared with type `integer`. Warning (not error) message is issued on double definition of variable.

10 New analyses and transformations

10.1 Induction Variable Recognition (IVR)

A new IVR recognition technique proposed by Michael Wolfe at paper "Beyond Induction Variables" published at PLDI '92 conference proceedings is implemented. This technique is based on Static Single Assignment (SSA) Graph. Right now we recognize only basic induction variables, that is, variables which are affine functions of loop parameters.

10.2 Forward substitution of scalars

This technique is based on the SSA graph too. We substitute variable by the expression assigned to it if this expression is considered to have *known* value. The following things are considered to have known value:

- Constants,
- Loop parameters,
- Result of operation which arguments are known values,
- Value of variable assigned with known value.

It should be noted that array reference, even if all of its indexes are known values, is not a known value.

10.3 Elimination of dead scalar assignments

Assignment to scalar is considered to be dead if value assigned by it is not used later in the program.

Decision to eliminate assignment is taken using the SSA graph of program. If there are no edges going from strongly connected component (SCC) in reverse of SSA graph (in reverse graph edge goes from definition of value to its use) then this SCC can be eliminated.

By eliminating not just single statement but whole SCC we are able to eliminate mutually dependent but otherwise useless assignments.

10.4 Example of SSA graph–related techniques application

```
real a(100,100,100),b(100),c(200)
integer n, s, t, u
real c1
s = 100
t = 1
for k = 3 to 100 by 2 do
  c1 = c(k)
  a(k,s+10,t) = a(k,s,t)+c1
  b(s+10) = b(s)+c1
  s = s - 10
  t = t + 10
  c1 = c(k)
  u = t
endfor
for k = 0 to 100 by 1 do
  b(t) = b(s)+1
  s = s - 10
  t = t + 10
  c1 = b(t)
  u = s
endfor
```

is converted to the following using IVR, forward substitution and dead statement elimination:

```
auto real a(1:100,1:100,1:100), b(1:100), c(1:200)
auto integer n, s, t, u
auto real c1
for k = 3,100,2 do
  c1 = c(k)
  a(k,125-k*5,-14+k*5) = a(k,115-k*5,-14+k*5)+c1
  b(125-k*5) = b(115-k*5)+c1
endfor
```

```
for k = 0,100,1 do
  b(k*10+491) = b(-390-k*10)+1
endfor
```

10.5 Reduction dependences

This is fourth type of dependence in addition to three known types — flow, anti, and output dependences.

There is a *reduction* dependence between two instances of statements iff these two instances may be executed in any order but not simultaneously or in such a fashion that the semantics of reduction is preserved.

Reduction dependences may exist only between variable updates of the same type (that is, there is a reduction dependence between `s+=expr` and `s+=expr` but not between `s+=expr` and `s*=expr`). Variable update is done by an update statement (see above).

Another distinctive feature of reduction dependences is that they are not forward in time. It is so because reduction dependences don't impose any ordering on sequence of commutative and associative updates of the same memory cell. They rather indicate that instances of statements can not be executed in parallel but it's OK to execute them sequentially in any order (or to use special reduction operations hardware support present in many parallel/vector computers).

```
real a(100,100)
real s

S4:  s = 2
     for i=1 to 100 by 2 do
     for j=3 to 100 by 3 do
S7:    s += a(i,j) + 2
S8:    s += a(j,i) - 1
     endfor
     endfor
S11: a(50,50) = s
```

In this example we have the following dependences coming from statement S7 (with Omega-4 turned off).

```
flow     7: s              -->  11: s
flow     7: s              -->  11: Exit
reduce   7: s              -->   7: s             (0,+)
reduce   7: s              -->   7: s             (+,*)
reduce   7: s              -->   8: s             (0,0)
reduce   7: s              -->   8: s             (0,+)
reduce   7: s              -->   8: s             (+,*)
```

Even though we say that reduction dependence is *between* statements S7 and S8, it is shown in Petit as going both from S7 to S8 and from S8 to S7 with the direction vectors splitted so that the dependence is forward in time.

10.6 Array and scalar renaming and expansion

This transformation allows us to kill anti and output (generic name — storage) dependences by renaming and expanding involved variables.

To kill storage dependence by expansion select menu entry `Browse.DD.Expnd` when dependence browser is on the dependence you want to kill. Please note that kill is not guaranteed because we use simplified algorithm for the expansion.

To run expansion for the whole program run menu entry `System.Auto.ExpanArr` or start Petit with command line flag `-x`.

We use the following renaming and expansion algorithm: For each variable

1 Find connected components of dependence graph based on flow and reduction dependences only. For each connected component:

2 To kill a storage edge between two different components, rename one of them.

3 To kill a storage edge within a component:
Find a depth d such that all flow and reduction edges within the component has distance 0 at level d and the edge being killed has a non-zero distance.
Note: all references must share the same outer d loops.
Rename the component, and add an additional subscript to all references, containing the loop variable for level d.

Example.

```
real a(1:100,1:100)
integer n
real c,s,d,a1,a2
for i = 1 to n do
  for j = i+1 to n do
    a1 = a(i,i)
    a2 = a(j,i)
    d = sqrt(a1*a1+a2*a2)
    c = a1/d
    s = a2/d
    for k = i to n do
      a1 = a(i,k)
      a2 = a(j,k)
      a(i,k) = c * a1 + s*a2
      a(j,k) = -s * a1 + c*a2
    endfor
  endfor
endfor
```

After scalar and array renaming and expansion:

```
inout real a(1:100,1:100)
auto integer n
auto real c_e(1:n,i+1:n), s_e(1:n,i+1:n), d_e(1:n,i+1:n)
auto real a1_e(1:n,i+1:n), a1_r_e(i:n,1:n,i+1:n)
```

```
auto real a2_e(1:n,i+1:n), a2_r_e(i:n,1:n,i+1:n)
for i = 1,n do
  for j = i+1,n do
    a1_e(i,j) = a(i,i)
    a2_e(i,j) = a(j,i)
    d_e(i,j) = sqrt(a1_e(i,j)*a1_e(i,j)+a2_e(i,j)*a2_e(i,j))
    c_e(i,j) = a1_e(i,j)/d_e(i,j)
    s_e(i,j) = a2_e(i,j)/d_e(i,j)
    for k = i,n do
      a1_r_e(k,i,j) = a(i,k)
      a2_r_e(k,i,j) = a(j,k)
      a(i,k) = c_e(i,j)*a1_r_e(k,i,j)+s_e(i,j)*a2_r_e(k,i,j)
      a(j,k) = -s_e(i,j)*a1_r_e(k,i,j)+c_e(i,j)*a2_r_e(k,i,j)
    endfor
  endfor
endfor
```

10.7 Array privatization

The basic idea of this transformation is to make each processor of MIMD system own separate instance of a given variable thus eliminating communication and subscripts computation. Privatized variables have storage class PRIVATE.

To apply privatization to the whole program select menu entry `System.Auto.Privatize` or start Petit with command line flag `-p`.

We use the following privatization algorithm: For each variable

1 Find connected components, just as in expansion. For each connected component:
2 For each component, let d be the level such that all flow and reduction dependences have distance 0 at this level.
3 Privatize the component at the level d. Put variable declaration inside the appropriate loop.
4 Remove any subscripts that are all identical (e.g., the variable is privatized in the `i` loop, and the second subscript is `i` in all references). Do not refer to unprivatized loop variables or any non-loop variables.
5 Repeat steps 2, 3, 4 until no new d can be found.

Example: after the privatization of previously expanded program:

```
inout real a(1:100,1:100)
auto integer n
for i = 1,n do
  for j = i+1,n do
    private real d_e_p, s_e_p, c_e_p, a2_e_p, a1_e_p
    a1_e_p = a(i,i)
    a2_e_p = a(j,i)
    d_e_p = sqrt(a1_e_p*a1_e_p+a2_e_p*a2_e_p)
    c_e_p = a1_e_p/d_e_p
    s_e_p = a2_e_p/d_e_p
```



```
    for k = i,n do
      private real a1_r_e_p, a2_r_e_p
      a1_r_e_p = a(i,k)
      a2_r_e_p = a(j,k)
      a(i,k) = c_e_p*a1_r_e_p+s_e_p*a2_r_e_p
      a(j,k) = -s_e_p*a1_r_e_p+c_e_p*a2_r_e_p
    endfor
  endfor
endfor
```

10.8 Data dependences from Entry and to Exit nodes

Entry node symbolizes the outer context of Petit procedure which *defines* variables of COMMON, STATIC, FORMAL, IN, INOUT storage classes.

Exit node symbolizes the outer context which *uses* variables of COMMON, STATIC, FORMAL, OUT, INOUT classes.

Dependences involving one of these nodes have empty direction and distance vectors because these nodes are not surrounded by any loops.

Flag `-g` makes Petit ignore dependences involving Entry and Exit nodes.

11 *f2p* — Fortran-77 to Petit converter

`f2p` is a hack of `f2c` (Fortran-77 to C converter) which, in turn, is a hack of ancient `f77` UNIX compiler. `f2p` is given name of Fortran program to convert:

```
f2p foo.f
```

If conversion is successful file `foo.t` will appear in your working directory. If you already had `foo.t` in your working directory it will be overwritten. Your favorite `*.t` files can be protected from overwriting by making them read-only.

If Petit is given a Fortran source, it automatically translates it to the Petit language by calling `f2p`. Since Petit doesn't know path to `f2p`, it should be on your PATH.

It's not recommended to use `f2p` on files containing more that 1 Fortran program unit (procedure or function). First use `fsplit` to make each file contain exactly one program unit.

If you try to convert something which can not be expressed in Petit, C code is likely to appear and to be rejected by Petit. `f2p` is not designed to convert full Fortran-77 to Petit. Moreover, it can not do it because of Petit restrictiveness.

Here's a short list of things that are converted but are not likely to be accepted by Petit:

- GOTO statements and any unstructured control statements. It may be recommended to use first some restructuring tool (like VAST, KAP, or even `struct`) to get rid of unstructured control.
- CHARACTER type and any manipulations with string values.
- RETURN statement in the middle of a program. In fact, RETURN statement is just special case of GOTO statement. So, if you have RETURN at the end of your SUBROUTINE, Petit

will accept it. But if you have it in the middle of a program, use third party restructurer first to make your program structured.

- COMPLEX things may cause a problem.

12 Syntax of Petit

This is taken from file `src/petity.y` which is YACC parser for Petit language.

Terminals are capitalized or are represented by input sequences in double quotes, non-terminals are lower case names.

```
Explanations for some of terminals:
TID             name
TINT            integer constant
TFLOAT          real constant
TEOS            ";" or "\n"  -- end of statement
TCOMMENT        comment starting with "!"


pgm:    stlist

stlist: stmt
|       stlist TEOS stmt

stmt:
|       stassert
|       stcomment
|       stasgn
|       stasgnoper
|       stconst
|       stfor
|       stforall
|       stif
|       stparfor
|       stdoany
|       stdecl
|       streturn

streturn:   "return"
|           "return" expr

stassert:   "assert" "(" expr ")"

stcomment:  TCOMMENT

stasgn:     lhs "=" expr
```



```
stasgnoper: lhs asgnoper expr

asgnoper:  "+="
|          "*="
|          "max="
|          "min="

lhs:       TID
|          TID "(" list ")"

stconst:   "const" constlist

constlist: constdecl
|          constlist "," constdecl

constdecl: newid "=" cexpr

stfor:     "do"  doid "=" expr forto expr optforby TEOS stmt
|          "do"  doid "=" expr forto expr optforby "{" stlist "}"
|          "for" doid "=" expr forto expr optforby "do" stlist "endfor"

optforby:
|          forby expr

stforall:  "forall" doid "=" expr forto expr optforby "do" stlist "endfor"

stparfor:  "doall"  doid "=" expr forto expr optforby "do" stlist "endfor"

stdoany:   "doany"  doid "=" expr forto expr optforby "do" stlist "endfor"

forto:  "to"
|       ","
|       ":"

forby:  "by"
|       ","
|       ":"

stif:   "if" expr "then" stlist elsepart "endif"

elsepart:
|       "else" stlist

stdecl: class type decllist
```

```
class:
|       "auto"
|       "common"
|       "static"
|       "formal"
|       "in"
|       "out"
|       "inout"
|       "private"
|       "builtin"

type:   "integer"
|       "real"

decllist: decl
|         decllist "," decl

decl:   newid init
|       newid "(" boundlist ")" init

init:
|       "=" expr

boundlist:
|       bound
|       boundlist "," bound

bound:  expr
|       expr ":" expr

newid:  TID

doid:   TID

cexpr:  TID
|       TINT
|       cexpr "+" cexpr
|       cexpr "-" cexpr
|       "-" cexpr
|       "+" cexpr
|       cexpr "*" cexpr
|       "(" cexpr ")"

oldid:  TID

expr:   oldid
```



```
|       oldid "(" list ")"
|       TINT
|       TFLOAT
|       "(" expr ")"
|       expr "+" expr
|       expr "-" expr
|       "+" expr
|       "-" expr
|       expr "*" expr
|       expr "/" expr
|       expr "**" expr
|       expr "<" expr
|       expr "<=" expr
|       expr "==" expr
|       expr "!=" expr
|       expr ">=" expr
|       expr ">"   expr
|       expr "mod" expr
|       expr "max" expr
|       expr "min" expr
|       TSQRT "(" expr ")"
|       "floor" "(" expr "/" expr ")"
|       "ceiling" "(" expr "/" expr ")"
|       assocop  "(" list ")"

assocop: "max"
|        "min"

list:   expr
|       list "," expr
```