Specification of **PARTITA** Analyses

Laurent HASCOET
Simulog Sophia-Antipolis

1 Introduction

The **FORESYS** system is devoted to understanding, cleaning, and transforming *FORTRAN* programs. All these tasks require deep and efficient preliminary analyses, in order to give acceptable results.

Most of these analyses are actually part of the **PARTITA** component of **FORESYS**, since they were initially devellopped for use by this component.

A key constraint in the design of **FORESYS** is that it must be used on real-life *FORTRAN* programs. This stresses the problems of efficiency, in time and space. This will often affect our implementation choices, sometimes leading to more complex algorithms. Also, since real-life *FORTRAN* programs may heavily use such concepts as `entry`, `exit`, `goto` or other unstructured control, all our analyses must perform reasonably well even in ustructured parts of a file.

Memory organization is complex in *FORTRAN*. It is a mix of old-fashioned `common`, `equivalence`, and of more modern `pointer` or `structure`. As a consequence, it is difficult and costly to determine in what manner two memory references may overlap. Since this question is crucial in most analyses, we have to design a way to answer it quickly. In the next section, we will present a solution based on a splitting of the memory space into atomic zones.

Control structures are complex in *FORTRAN*, since many programs were built with gotos, *i.e.* in absolutely free control. This leads to pathologic cases, such as irreductible loops. Moreover, some forbidden programming methods were not checked for by old compilers, and therefore were widely used. Even if a goal of the **FORESYS** system is to detect and remove these cases, we must first analyse them correctly. As a conclusion it turns out that the best program representation for our purpose is not the abstract syntax tree, but the control flow graph (*cfg*). The *cfg* will be explained in detail in the 4^{th} section. All the analyses that we shall present later, in the 6^{th} and 7^{th} sections, are based on the *cfg*.

The analyses we shall present here run on a single *FORTRAN* `unit`. Then it is the task of the interprocedural database system to combine the analyses of these **units**, so as to get a correct interprocedural result. We shall not talk of these interprocedural issues here.

Let us present the analyses that we implement, in a "call by need" fashion. There are two main results we want to provide the user. One is the information about which variables are used, overwritten, or both, by the considered unit. We call this the **in-out analysis**. The other result is the **dependence graph**, linking all the basic operations of a unit with dependence relations. These relate reads and writes of a same value (**data dependencies**), or control structures with the operations that they condition (**control dependencies**).

As is well known, a good dependence graph must contain as few dependencies as possible. To achieve that, array references must be carefully studied to detect that array indexes may never overlap. Classical analyses help detect these independencies. For example, knowing the interval of value of a variable may help proving that two array indexes never overlap. Also, knowing that a variable is an arithmetic progression with respect to the loop index, helps too. Also, some directives provided by the user must be propagated to every place where they might be useful. For the control dependencies, a careful analysis is needed to limit the scope of influence of conditionnals and loops. For the in-out analysis, an important improvement is the detection of killed arrays, i.e. arrays which are proved to be completely overwritten through the execution of a loop.

To summarize, the analyses that we need for **PARTITA** and **FORESYS** are

- **variable bounds:** Searches and propagates information about the interval inside which the run-time value of each numerical variable will range.
- **killed arrays:** Detects the array variables whose value is completely overwritten through the execution of a loop.
- **in-out analysis:** Computes, for each variable of a program unit, whether it is read, or overwritten, or a combination of those, through the execution of the unit.
- **context of control:** Computes, for every line of the unit, upon which conditions and inside which iterations it is executed.

- **induction variables:** Detects these variables whose value can be proved a simple arithmetic progression with respect to the loop control index.
- **directives propagation: PARTITA** allows the user to give directives. Then this analysis propagates these directives through the unit, until some operation makes this directive obsolete or false.
- **data dependencies:** Detects all possible data dependencies between every couple of memory references found in the unit. Three types of data dependence exist: *write-to-read*, *read-to-write*, and *write-to-write*.

There is an order between these analyses, since the result of one may be useful to another. This order is shown on figure 1.

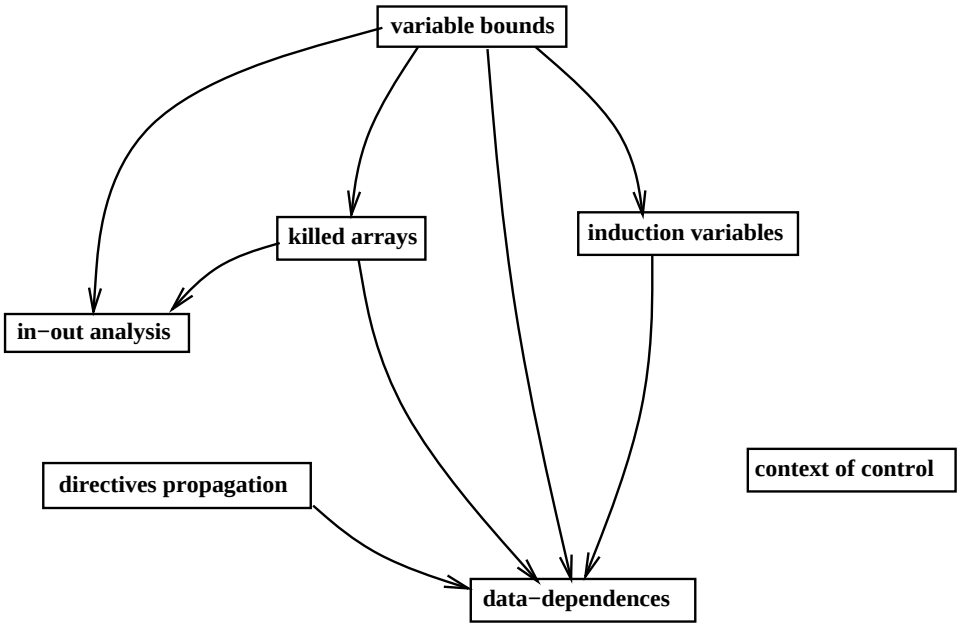

Figure 1: *Ordering of the current* **PARTITA** *analyses*

There are cases where two different analyses may benefit from each other. Since this leads to a circularity, we choose to loose the benefit in one direction, thus loosing some precision in the analysis. For example, a good detection of induction variables needs to know the bounds inside which a loop index ranges, but conversely the bounds of a variable are more precisely known when it is proved to be an induction variable. One solution would

be to interleave these analyses, but this would lead to extremely complex algorithms.

All these analyses are based on the *cfg*. But, except for the **context of control** and the **directives propagation**, they go deep inside each *FORTRAN* instruction, to the level of memory references and atomic operations. This introduces a second graph, which we call the data flow graph (*dfg*), and which represents these memory references and atomic operations. Besides, this *dfg* is also the place where data dependencies will be stored. The *dfg* will be described in the 5^{th} section.

2 The memory map

Because of equivalences, variables of different names may overlap in memory. It is important to consider that when computing the **data dependencies**. Similar problems come with **pointers**, or with *aliasing*.

Checking that two variable references overlap may be expensive if we always get down to comparing their memory offset. So there is a preparation phase, during which the memory space is divided into zones. A given memory reference corresponds to one or more zones. Most computation during analysis refer explicitely to zones instead of variable names.

The zones are chosen so that it is easy to tell two zones overlap. Actually, two zones overlap if and only if they are the same, *i.e.* they have the same rank.

Formally, the zones compose a partition of the memory space. It is the largest partition such that if a variable name may access that zone, it may access every byte of it. In other words, no beginning or end of an array (or a scalar) may be in the middle of a zone. Figure 2 shows an exemple of zones decomposition, in the case where A(100) has been equivalenced with B(1), and B(250) has been equivalenced to scalar X.

There is another decomposition of memory space, called "big Zones" or "Zones", also shown on figure 2. Big Zones are designed so that no variable can be in two Zones. This is used in **data-dependencies** only, and will be explained there.

The preparation of the **memory map** yields the list of all zones. Every zone knows its offset, and the list of all variables that may access it. Conversely, every variable reference will be given a **varinfo**, which contains information about the accessible zones.

To save memory space, most **varinfo** are shared between all variable references of same name. They are thus computed during **memory map**

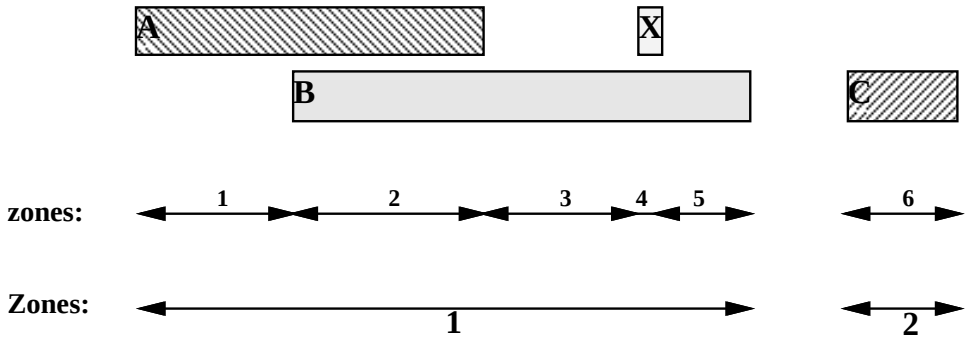

Figure 2: *zones and Zones of the* **memory map**

creation, stored in the symbol table for their variable name, and later (during *dfg* creation), each variable reference will receive its **varinfo**. An exception is **pointers**, that require pointer-based variables to have possibly different **varinfo**'s at each occurence.

This **memory map** is slightly different to handle data structures, but we shall not discuss this here.

3 Graphs

There are many classical ways to represent a graph. We did not elect the matrix representations, devoted to dense graphs, since both *cfg* and *dfg* are sparse graphs. Moreover, we want enough flexibility to add extra nodes whenever we want.

So the representation we chose is a list of nodes, and each node stores the list of the arrows related to it. A question remains: should a node keep the arrows flowing to it, from it, or both? This is a classical trade-off, whose answer is different for *cfg* and *dfg*.

The *cfg* is relatively small, and has few arrows. about half of the analyses will traverse it from the entry down, and the other half from the exit up. It is natural to store the control flow arrows in both directions.

The *dfg* is very large. It has nearly as many nodes as machine instructions contained in the **unit**. Even with a very good data dependence analysis, the number of data dependencies is high. Moreover, it turns out that nearly all analyses use only one direction of the arrows: a *dfg* node needs only know about the data dependencies going *to* it. It is therefore natural to store only the *dfg* arrows arriving at a given node, and this saves memory space.

As we said before, the analyses traverse the *cfg*, but use the *dfg* at the same time. It is actually better to think of these analyses, not as *cfg* or *dfg* based, but based on an intimate combination of both graphs. For this reason, these two graphs are stored together, in a structure called *flow-graph*, and constantly reference each other. For sake of clarity, we shall try to present them separately, in the two next sections, but this is an artefact.

4 The *control flow graph*

The *control flow graph* is a classical representation of imperative programs. It consists of *basic blocks*, which are sequences of *atomic instructions* that are always all executed in order. *Basic blocks* are then linked with *control flow arrows*, that indicate the possible choices for the control flow, to go from a *basic block* to the next one to be executed.

Atomic instructions may be an assignment, a procedure `call`, an io statement, a `do` header, a conditionnal test, by opposition to *structured instructions* such as a `do` or an `if then else`.

In the following, basic blocks will be called block-node's (structure **block-node**), the control flow arrows are instances of the structure **c-arrow**, and atomic instructions are instances of the structure **instr-node**.

There are two special nodes of the *cfg*. One is the entry block, of subtype **block-node:entry**. It corresponds to the beginning of the unit. Each instruction of this block corresponds to a different `entry` statement, corresponding to the various ways to call the unit. The first of these instructions corresponds to the normal entry, *i.e.* the header of the unit.

The other special node is the exit node (subtype **block-node:exit**), corresponding to the normal end of the unit, returning control to the calling unit. It has only one **instr-node** corresponding to the "end" statement.

Among all the other nodes of the *cfg* are nodes of subtype **block-node:header**. They represent loop headers, *i.e.* the first basic block of a loop, when the loop is entered normally. Basic blocks of type **block-node:header** are special since many extra information concerning the loop itself will be stored there. See the annex for a complete list of those.

Since loops in a *FORTRAN* program are not always signalled by a `do` statement, we must detect all loops directly from the *cfg*. This is done with a classical algorithm, a *depth first search* in the graph, starting from the entry block. An extension of this algorithm finds the loops in the *cfg*. This extension is very close to the classical *Tarjan algorithm*. In the rare (and disapproved) case of a loop with many entries, if one entry corresponds to

a `do` loop, we ensure with a backtracking mechanism that this entry will be chosen as the **block-node:header**.

A limitation inherent to the algorithm is that loop detection associates one loop per loop header, i.e. it is not able to split a loop in two nested loops, when the two loops cycle through the same loop header. This would be useful in some cases, but it cannot be achieved automatically and would require an interface with the user.

An interesting by-product of the *depth first search* is that it returns an ordering of the **block-node**'s. This ordering called the *dfst* order, is such that every propagation of a value through the *cfg*, in the control flow direction, is faster when **block-node**'s are visited in *dfst* order. When the propagation goes backwards, just take the nodes in the reverse of *dfst* order.

Our *cfg* creation algorithm also performs some simplifications. One is the detection of dead code. Precisely, we detect and remove from the *cfg*, all **block-node**'s that cannot be reached from the entry node. This dead code is stored in the **flow-graph**. Also, we minimize the size of the *cfg* according to the following rules:

- Empty **block-node**'s must be skipped
- **block-node**'s that are always consecutive, the first always flowing to the second and the second only reached by the first, must be merged.
- **c-arrows** going from the same origin **block-node** to the same destination **block-node** must be merged.

The assigned `goto` statements are partly solved at *cfg* building time, *i.e.* we propagate the values that a goto variable may have, and set only those *c-arrow*'s from the goto statement to the *block-node*'s labeled by a possible value of the goto variable. Error messages are emitted when an illegal use of a goto variable is detected.

Figure 3 shows the *cfg* built for a small **unit**.

4.1 The block-node's

The **block-node**'s correspond to basic blocks. A **block-node** must thus contain the ordered list of its atomic instructions, or **instr-node**'s. It also contains the list of the **c-arrows** flowing towards it, and from it. There is also an indication about the immediately enclosing loop header, if any. The **block-node** holds additional information, returned by the various analyses. See the annex for a complete list.

```
      subroutine small(A,B)
      integer A,B,i,n
      dimension A(100),B(100)
C
      n = 0
      do 100 i = 1,100
        if (A(i).ge.0)
          then
            A(i) = n
            n = n + 1
          else
            A(i) = B(i)
            goto 200
            n = n + 3
        endif
        B(i) = 0
 100  continue
 200  print *, n
      end
```

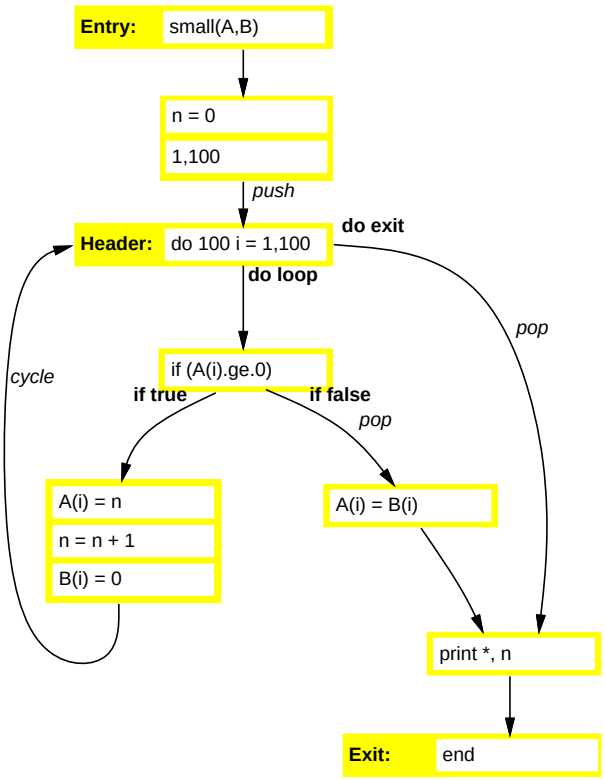

Figure 3: *Control flow graph* for a small **unit**

4.2 The block-node:header's

In addition to the contents of a **block-node**, the **block-node:header**'s contain specific information about the loop they represent. Let us mention here the list of all the **block-node**'s that are directly inside the loop, and not inside sub-loops. This list is stored in *dfst* order. See the annex for a complete list.

4.3 The instr-node's

The **instr-node**'s correspond to atomic instructions. They contain the abstract syntax tree of their instruction, plus a pointer to their enclosing **block-node**. In the case of conditionals and `do`'s, the abstract syntax tree is the whole structured statement, even if it represents only a part of it. **instr-node**'s also hold additional information, returned by the analyses.

Conversely, there is a way to go from the abstract syntax tree of an instruction to its corresponding **instr-node**.

4.4 The c-arrow's

The arrows of the *cfg* link the end of a *basic block* to the beginning of another. They express the fact that the control flows from the last instruction of the first block to the first instruction of the last block. When many arrows are available, they must tell in what case they are chosen. The switching instruction, that determines in what case we are, is always the last **instr-node** of the origin **block-node**. A **c-arrow** contains:

- its origin **block-node**
- its destination **block-node**
- the condition upon which it is chosen (its **type**)
- the loops it exits and enters
- the value to propagate along the arrow

The condition upon which an arrow is chosen (its **type**) is unnecessary when there is only one exiting arrow. In other cases it is the conjunction of a **typename** and a list of **cases**. Sometimes the list of **cases** may be replaced by "default", meaning all the **cases** not taken by other **c-arrows**. Possible **types** are shown on figure 4

typename	case	default
comp_goto	*n*, positive integer result of the expression	*yes*
select_case	the syntax tree of the case selector	*yes*
io	*end*, *error*, or *normal*	*no*
ass_goto	the target goto label	*no*
call	the alternate return label, or *normal*	*no*
if	*true*, *false*, *lt*, *gt*, *le*, *ge*, *eq*, *ne*	*no*
entry	the entry name or *%main%*	*no*
do	*loop* or *exit*	*no*

Figure 4: Possible types of a **c-arrow**

The loops that the arrow exits and then enters are stored in an ordered list. It starts with the (maybe empty) list of all the loops exited, then either there is a single loop name into which the arrow cycles, *i.e.* the control flow starts a new iteration, or there is a (maybe empty) list of all the successive new loops into which the control flow enters. The loops are represented by their **block-node:header**'s BH_i, thus this info has one of the two shapes:

- pop BH_1, then pop BH_2, ... then cycle BH_n
- pop BH_1, then pop BH_2, ... then push into BH_n, then push into BH_{n+1}, ...

4.5 The two ways to traverse a *cfg*

Analyses of a *FORTRAN* **unit** are based on a traversal of the *cfg*. There are two main ways to do so. According to the analysis, the best one must be selected. The *cfg* may be traversed:

- In a pure graph-oriented way, from the entry down or from the exit up. In that case, there must be a fixpoint, since the graph contains loops, so that values are not guaranteed to be propagated everywhere in one pass.
- In a loop-oriented way. In that case, each loop is considered as a single instruction for its enclosing loop. At the end of each loop treatment, the result must be summarized for use by the enclosing level. There may be no need for a fixpoint then.

Let us sketch here the framework of a graph-oriented analysis, performed from the entry down. The aim is to propagate information on the *cfg*,

following the **c-arrow**'s, until this propagation yields nothing new, i.e. a fixpoint has been reached. Briefly the method is:

```
Initialize values of all cfg block-node's
Until <nothing has changed>
  {<nothing has changed> = true
   for all <block-node> in cfg, taken in dfst order
     {<last-value> = value propagated on previous passage
      accumulate the values coming from arriving c-arrow's
      compute the accumulated value
      <value> = propagation of this value across <block-node>
      propagate <value> onto exiting c-arrow's
      unless <value> == <last-value>
        {<nothing has changed> = false}
     }
  }
Terminate, and clean values of all cfg block-node's
```

This method is slightly improved in real implementation, to avoid memorizing all the values for all the *cfg*, and for detecting fixpoints earlier, thus saving redundant computation.

Besides these optimization issues, the main questions to answer to implement an analysis are now:

- What data structure to represent the values. Special attention must be paid to the representation of the two values: "unknown value", and a special value $\perp$ meaning that the **block-node** has not been visited yet. These are of course two *different* values.
- How to combine two values arriving from two converging arrows?
- How to compose the value upon entry into a **block-node** to get the value upon exit (or conversely)?
- How to tell two values are the same?

5 The *data flow graph*

The *data flow graph* (*dfg*), represents the dependencies between machine atomic operations, such as load, store, arithmetic operations, etc... Since the *dfg* does not expand function and procedure calls, these calls are also

considered as atomic operations. Other atomic operations are **io** operations, and `entry` and `exit` of the **unit**.

Since the *dfg* is very large, only incoming dependencies are stored on the nodes. As a consequence, arrows need not store their destination node, but only their origin, and their type.

5.1 The *dfg* nodes

These nodes will be implemented as subtypes of the type **op-ref:atomic**. This fundamental type has methods to get/set the list of incoming data-flow arrows, and to get/set the abstract syntax tree corresponding to the machine atomic operation. There is also a way to get the **instr-node** it comes from.

The *dfg* nodes that represent a load or a store are of subtype **op-ref:atomic:rw:read** or **op-ref:atomic:rw:write**, and have special methods to get the **varinfo** attached to the variable name (*cf* section 2). There are also additional methods used during **data dependencies** computation.

The subtype **op-ref:atomic:eval** represents operations, and the subtype **op-ref:atomic:constant** represents immediate constants. There is also a special **op-ref:atomic:loop-counter** attached to every loop, and representing the counter of iterations of the loop. The first iteration is numbered 0, the next one 1, and so on.

The *dfg* nodes are created by a simple analysis of each **instr-node**. They are then arranged, not as a flat list, but as a nested structure that reflects their relations in the instruction. This structure will be called a *nest*. The *nest* structure looks very much like the abstract syntax tree structure, except that its leaves are the **op-ref:atomic**'s, and it is implemented more lighly. The nest is then attached to its **instr-node**. For example, figure 5 shows the *nest* containing all **op-ref:atomic**'s for the instruction:

```
A(2*i) = B(i) + func(i+10,X)*C(i)
```

Since some analyses need to consider the excution order of all the **op-ref:atomic**'s, we provide a function that returns such an order from the *nest* structure. However, many orders are available. For instance, the *FORTRAN* standard give no indication on the order of evaluation of the arguments of a sum, or a product, or the arguments of a function call. We detect the situations where the result of an evaluation may change depending of the above unknown orders, and issue a warning message.

There are special **op-ref:atomic**'s created for the entry inside the **unit**, representing the write of the initial value of all formal parameters and commons, or the initial write of variables in `save` or `data` statements. On the other hand, for the exit from the unit, there are **op-ref:atomic**'s created for

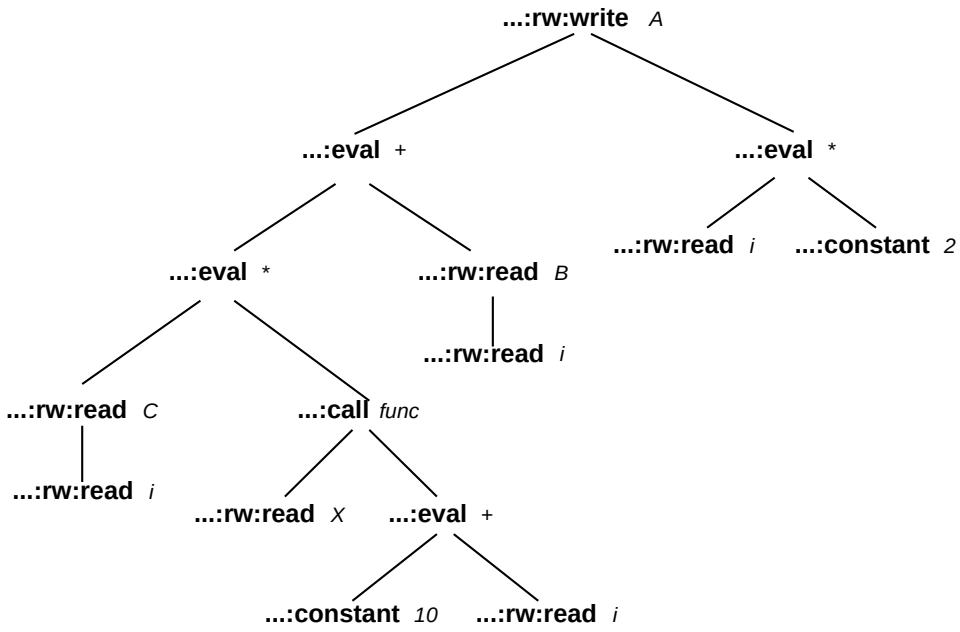

Figure 5: *nest* of all **op-ref:atomic**'s of an instruction

the read of the exit values of the `parameter`'s, `common`'s, and `save`'s for use by the calling **unit**. The **op-ref:atomic**'s on entry all depend on a single **op-ref:atomic:entry**. The **op-ref:atomic**'s on exit all have a dependence towards a single **op-ref:atomic:exit**

5.2 The *dfg* arrows

These arrows, of type **d-arrow**, represent all dependencies. The dependencies we want to find in the final *dfg* are:

- The *direct dependencies.*
  This is the dependence from the **op-ref:atomic:rw:write** of a memory location to a **op-ref:atomic:rw:read** that may access the value written at this same location.

- The *anti dependencies.*
  This is the dependence from the **op-ref:atomic:rw:read** of a memory location to a **op-ref:atomic:rw:write** that may overwrite the value read at this same location.

- The *output dependencies.*
  This is the dependence from the **op-ref:atomic:rw:write** of a memory location to a **op-ref:atomic:rw:write** that may overwrite the value written at this same location.

- The *control dependencies.*
  This is the dependence from the top **op-ref:atomic** of a control statement (loop or conditionnal) to every **op-ref:atomic** of instructions actually controlled by it.

- The *value dependencies.*
  This is the dependence from an **op-ref:atomic** returning a value on the execution stack to the **op-ref:atomic** using this value.

A **d-arrow** memorizes all the dependencies from its origin **op-ref:atomic** to its destination **op-ref:atomic**. It also stores its origin **op-ref:atomic** itself, but not its destination.

It is well known that all these dependencies have a *distance*. This is the difference between the loop iteration where the origin **op-ref:atomic** is executed, and the loop iteration where the destination is executed. This distance is essential for parallelization analyses.

6 The analyses performed on *cfg* and *dfg*

6.1 The variable bounds analysis

The computation of variable bounds is done by a fixpoint traversal of the *cfg*, from the entry down. As we saw before, all we need to specify now is the structure used to keep the bounds, then the way two such structures combine (when two **c-arrows** converge), the way this structure is updated through a **block-node**, and how to detect that the fixpoint is reached.

As one would expect, bounds are computed zone per zone. We limit the expressive power of our variable bounds, by associating only an interval of real values to each zone. This way we are not able to use and propagate constraints such as $x \geq y$ or $x \neq 0$ The structure used is therefore a vector associating to each zone number an interval of real values, extended with $+\infty$ and $-\infty$. It may also have a special *impossible* value. When a arrow carries the *impossible* value, it means that this arrow is never taken by the control flow. If all arrows arriving to a **block-node** have this value, it means that the **block-node** is dead code.

The way two variable bounds vectors combine, coming from two converging arrows, is simple. We just compute the interval union of the incoming value intervals.

The way a variable bounds vectors is modified by an instruction is just abstract interpretation. Abstract interpretation of the left hand side of an assignment returns an interval (maybe $[-\infty, +\infty]$). Then the zones of the right hand side receive this value. When the write is *total*, i.e. all memory cells of the zone are surely written, then the interval overrides the previous one. When the write is not *total*, i.e. the write is not certain, or it does not override all cells of the zone, then the interval is merged with the previous one.

Moreover, all instructions, and not only assignments, are searched for side-effects, for instance in function `call`'s. These side-effects may override the value of some zones with an unknown value. Therefore, the variable bounds vector must be updated in consequence, setting $[-\infty, +\infty]$ into the corresponding zones.

We have a special way of detecting fixpoint. On each **block-node**, we remember the bounds vector at **block-node** entry, at previous traversal. We do not compute the same bounds vector at current traversal, but increase the previous bounds vector. This is natural since, by definition, bounds contain a summary of the variable bounds in all cases, and therefore, exploring new cases can never return smaller bounds. In other words, bounds vectors are strictly growing during bounds computation. Fixpoint is easily found, when no zone has been assigned a larger interval.

Lastly, let us mention three important characeristics of the algorithm:

- In loops, one may found an infinite sequence of growing bounds. This happens for instance when x is increased by 2 at each loop. We may never reach the fixpoint. To avoid this, there is a mechanism of *widening*. The bounds interval on iteration n is checked with the one on iteration $n-1$. If it has increased towards $+\infty$, then it is widened by replacing the upper bound with $+\infty$. The same applies for $-\infty$.

- Conditionnals are used to get more information, according to which exit arrow is taken. If we are on the false branch of the test:

  ```
  if (X.ge.10 .or. Y.lt.0)
  ```

  we may refine the bounds, knowing that X is less than 10 and Y is positive. If the bounds on one arrow has the *impossible* value, then this means that the corresponding branch is dead code.

- Additional information is also extracted from `data` statements, as well

as from a category of user's directives, specifying arithmetic properties of variables. If the directives clearly contradict what was known before, a warning message is emitted, and the directive is given priority.

6.2 The killed arrays analysis

The **killed arrays** analysis is done with a top-down traversal of the tree of nested loops. At each level, the block-nodes contained in the loop are traversed in *dfst* order.

To speed up the analysis, there is a preliminary phase of detection of the *killable arrays*, *i.e.* the arrays that are likely to be killed by some loop, and the *killer loops*, *i.e.* the loops that are simple enough so that one can prove that they kill an array.

The *killable arrays* are the arrays that are local parameters, or that are in a common. Also, they must not be of "assumed size", *i.e.* their size must be known statically.

The *killer loops* are the do loops such that:

- there is no other entry into the loop but the normal one.
- there is no other exit from the loop but the normal one.
- the loop bounds are static constants (maybe proved by **variable bounds** analysis.
- the loop stride is either 1 or -1

Also, during the analysis, only the *good writes* will be considered for array kill. A write of an array is *good* iff for every array dimension, the index expression is a constant or a linear function of the index of a *killer loop*. So far, these *good writes* are not all useful for **killed array** analysis, but they might be used by a future better algorithm.

The value propagated during traversal of the *cfg* is the following one:

For each *killable array* **A**,
For each *good write* of **A** encountered so far,
For each dimension of **A**,
- Either a constant
- Or a linear function of some *killer loop* index.

The way this value is propagated is the following:

- Upon entry into a **block-node**, the incoming values are merged, keeping only their intersection, i.e. a *good write* is kept iff it can be found in each incoming value.
- During traversal of a **block-node**, when a *good write* is met, it is added into the value.
- Upon exit from a **block-node**, the value is propagated on each exiting **c-arrow**

At the beginning of each loop, the value just before the loop is kept. A new, empty value is built, for propagation into the loop.

At the end of each loop, the values on all *cycle* arrows are merged. The result is the collection of all the *good writes* that are performed at each loop iteration, whatever the flow of control for this iteration. This value is analysed with respect to the current loop's bounds, and yields the names of the arrays effectively killed. This is the final result of this **killed arrays** analysis.

This value is also added to the value just before the loop, for propagation inside the enclosing loop.

6.3 The in-out analysis

The goal of **in-out** analysis is to determine, for each variable, whether it is read, written, or both, during an execution of the unit. This analysis is implemented with a fixpoint traversal of the *cfg*, from the exit up.

For a given zone, its read status can be true (**t**) or false (**/**). There is a third status, called "maybe" (**?**), which represents the case where the zone is maybe read, maybe not, depending on the control flow. This **?** also represents the case of the zones that are partly read (case of arrays).

Similarly, the write status has also three possible values, **t**, **/**, and **?**. The read-write status describes the way the initial value of the zone is read, then the way it is overwritten later. It makes no sense talking of how this value may be read *after* it was overwritten, precisely because it was overwritten! The read-write status of a zone has thus nine possible values (read status first):
//, **/?**, **/t**, **?/**, **??**, **?t**, **t/**, **t?**, **tt**

The structure to propagate along the *cfg* is therefore a vector, associating each zone with its read-write status. It represents, for each zone, the accumulated read-write status from the current point in the *cfg* to the exit of the *cfg*, taking into account every possible control flow.

The structure is actually more complex since it must also keep and update read-write information for variables that have no zone in the current unit, since they appear only in called subroutines or functions, and communicate through `common`'s.

At initialization time, each **block-node** is provided with the read-write status through it. This will speed up later propagations. This is natural since a **block-node**, having no switch in it, always affects the read-write status in the same way. The read-write status of a **block-node** is computed by retrieving all the **op-ref:atomic**'s of the *nest* of each **instr-node** of this block, in execution order, and then combine them in "then" mode (*cf* later). The read-write status of a subroutine or function call is given by the interprocedural analysis.

During the fixpoint, there are two improvements to the natural propagation.

- Loops that have certainly at least one iteration are treated in a special way. This is because, if a read or write occurs inside a loop that may be done zero times, then the status must be "maybe". On the other hand, the status may be true when one is certain the loop body is done at least once. This refinement uses the results of the **variable bounds** analysis.

- Arrays killed by a loop have a write status that is certainly true, even if the simple propagation returned "maybe".

Then, the last thing to say to fully specify the **in-out analysis** is how two read-write status combine, in the *or* mode (when two **c-arrow**'s converge), and in the *then* mode (when the two status are done in sequence). This is summarized in Figures 6 and 7.

The Final result of **in-out analysis** is used in the following way. It is split in two. One part is the in-out information for variables that are exported, *i.e.* the variables that are (equivalenced to) formal parameters or in `common`'s. The rest is the local variables information.

The first part is stored into the interprocedural database to be used by **in-out analysis** of calling units. It can also be shown to the user *via* the **FORESYS** interface.

The second part is scanned to detect those zones whose read status is true. This corresponds to an error, since it means the zone is used before its initialization by the user. In all other cases, there is nothing to say. This second part is not kept after **in-out analysis**.

	//	/?	/t	?/	??	?t	t/	t?	tt
//	//	/?	/?	?/	??	??	?/	??	??
/?	/?	/?	/?	??	??	??	??	??	??
/t	/?	/?	/t	??	??	?t	??	??	?t
?/	?/	??	??	?/	??	??	?/	??	??
??	??	??	??	??	??	??	??	??	??
?t	??	??	?t	??	??	?t	??	??	?t
t/	?/	??	??	?/	??	??	t/	t?	t?
t?	??	??	??	??	??	??	t?	t?	t?
tt	??	??	?t	??	??	?t	t?	t?	tt

Figure 6: The *or* combination of read-write status

	//	/?	/t	?/	??	?t	t/	t?	tt
//	//	/?	/t	?/	??	?t	t/	t?	tt
/?	/?	/?	/t	??	??	?t	??	??	?t
/t	/t	/t	/t	/t	/t	/t	/t	/t	/t
?/	?/	??	?t	?/	??	?t	t/	t?	tt
??	??	??	?t	??	??	?t	??	??	?t
?t	?t	?t	?t	?t	?t	?t	?t	?t	?t
t/	t/	t?	tt	t/	t?	tt	t/	t?	tt
t?	t?	t?	tt	t?	t?	tt	t?	t?	tt
tt	tt	tt	tt	tt	tt	tt	tt	tt	tt

Figure 7: The *then* combination of read-write status (line *then* column)

6.4 The context of control analysis

The aim of the **context of control** analysis is to determine, for each **block-node**, by which loops and by which conditionnals it is controlled. Actually, a **block-node** may depend on a loop to know how many times it will be executed (maybe zero), and with what values of the loop index. Similarly, a **block-node** may depend on a conditionnal to know if it is executed or not.

If the original program was structured, these questions would be trivial. However, in *FORTRAN*, unstructured instructions lead to surprising cases. For example, a **block-node** inside a `do` loop may not be controlled by that loop, because it is done only before an `exit` instruction.

6.4.1 obliged successors

One preliminary analysis is to compute, for each **block-node**, the list of all its *obliged successors*, *i.e.* the list of all **block-node**'s that will be executed after it, whatever the control flow.

This is done by a single traversal of the list of all **block-node**'s, in the reverse *dfst* order. It turns out that this order avoids the fixpoint.

The propagated value is a pair of two lists. One is the list of the **block-node**'s that are obliged successors of the current **block-node**, in the whole program, the other is the list of the obliged successors, *inside the current iteration* of the enclosing loop. The general idea is that the obliged successors of a **block-node** are the intersection of the obliged successors of the **c-arrow**'s flowing from it. The obliged successors of a **c-arrow** are the obliged successors of its destination **block-node**, *plus* the destination **block-node** itself.

6.4.2 context of control

This is computed by a single traversal of the list of all **block-node**'s, in the *dfst* order. Each **c-arrow** conveys the **context of control** for which it is taken. Each **block-node** computes its **context of control** by computing a logical **OR** between all its incoming **c-arrow**'s.

The key is the frequent simplification of the **context of control**. The main simplification is when a **block-node** is in the *obliged successors* of a conditionnal, then one may remove every reference to this conditionnal in its **context of control**. Other simplifications are the usual logical simplifications to normal form. We try and simplify out the **OR** logical operation as much as possible.

For controls with respect to loops, the following conventions apply:

- A **block-node** directly inside a loop is controlled by its loop header. See later for the syntax of **context of control**'s
- When a loop has many exits, then inside its enclosing level, it is considered as a switch controlling those **block-node**'s that are executed after one `exit` and not after another.
- When a loop has many entries, it is the loop header that has a complex **context of control** with respect to the different entries, while the blocks inside the loop have a simple control with respect to the loop header.

At the end, every **block-node** knows its **context of control**, whose syntax is:

`<context> ::= <condition>*`

i.e. a context is a list of conditions that must be all true, for the **block-node** to execute. These conditions are stored in reverse chronological order, the first condition to check is the last in the list.

`<condition> ::= "or" <context>*`

i.e. a condition may be the "or" conjunction of some contexts. This is the case where two flow arrows lead to a place, and they could not be merged into a single context.

`<condition> ::= "loop" <block-node:header>`

This means that the current place is conditionned by the iterations of the loop `<block-node:header>`.

`<condition> ::= "pop" <block-node:header> <rank>`

This means that the current place is conditionned by the way loop `<block-node:header>` exited. If it exited through the **block-node** of rank `<rank>`, then the condition is satisfied.

`<condition> ::= "push" <block-node:header> <rank>`

This is used only for the loop headers. This means that the current loop `<block-node:header>` will be entered through the push **c-arrow** from the **block-node** of rank `<rank>`

`<condition> ::= "test" <block-node> <c-arrow's type>`

This means that the current place is conditionned by the result of the test at the end of `<block-node>`. When the **c-arrow** chosen has the required type, then the condition is satisfied.

These results are immediately used at the end of **context of control** analysis, to set the **d-arrow**'s in the *dfg* corresponding to these controls:

- From the **op-ref:atomic:eval** of a conditionnal to every **op-ref** contained in a controlled **block-node**.

- From the **op-ref:atomic:eval** of a conditionnal to the **op-ref:atomic:loop-counter** of a loop, in the case where the test may determine an `exit` from this loop.

There should also be **d-arrow**'s from the **op-ref:atomic:loop-counter** of a loop to the **op-ref**'s contained in the loop. However, we do not set them, and will set only a few of them later. The reason is that we shall later detect the parts of the loops that are always the same at each iteration. These parts should be done only once in a restructured code. In that case, there must not be any dependence from the **op-ref:atomic:loop-counter** to these places.

6.5 induction variables analysis

The **induction variables** analysis detects which scalar variables may be expressed as linear functions of the loop indexes of enclosing loops.

The method we use here is implemented as a traversal of the tree of nested loops. Inside each loop, the **block-node**'s are traversed in *dfst* order. For each loop, there are three main steps:

- The first step is done during traversal of the **block-node**'s of the loop. We consider one loop iteration. Suppose we know ZV_n, the value of each zone at the beginning of this iteration. What we propagate is, for each zone, the way its current value is related to ZV_n. We propagate only the values that will allow us to find induction variables, i.e. polynomes of zones of ZV_n. In all other cases, the value propagated for the zone is a special constant, called *lost-track*. When the control flow merges, the polynomes on each incoming **c-arrow** must be the same, and then it is the merged value. Else the merged value is *lost-track*.

- The second step is the *closure step*. We compute ZV_{n+1} by collecting the values propagated on the cycling arrows of the loop. Then we analyse ZV_{n+1} as a function of ZV_n, to find out induction values:

  - if $z_{n+1} = c$ then z is a loop constant. $z_n = c$.
  - if $z_{n+1} = z_n + c$ then z has an induction value, which is $z_n = z_0 + c * lc$, where lc is the loop counter.
  - if z_{n+1} is a sum of zones that have already an induction value, then z has an induction value too.

- At the end of each loop, when the loop and the loop bounds are simple enough, we compute the exit value of the induction variables. This is useful to detect when the zone has an induction value with respect to the enclosing loop too. This will also generate more elegant code when some code transformation techniques require that the exit value of induction variables be set explicitely on exit of the loop.

6.6 directives propagation analysis

The only directive we need to propagate is the `C$injective` directive. All the others directives have already been used, during *dfg* building and **variable bounds** analysis. However, new directives may come up some day, that sould be propagated here.

This analysis is again implemented with a fixpoint traversal, from the entry down, of the *cfg*. There is no subtelty worth describing. The propagated value is the current list of injective array names. An array becomes injective when we reach a `C$injective` directive for it. An array remains injective as long as it is not overwritten. At a merge point in the *cfg*, an array must be injective on all merging arrows to remain injective.

7 The data dependence analysis

The **data dependence** analysis must perform a two-level traversal of the *cfg*. The first level chooses every possible origin **block-node**, and then for each **block-node**, the second level traverses the *cfg* to reach every possible destination **block-node**. This is why this analysis is grossly $O(N^2)$ where N is the number of **op-ref:atomic:rw** of the **unit**. This preamble is to justify all the efforts to speed this algorithm up, since it consumes most of the time spent by **PARTITA**.

There are three successive granularity levels in the algorithm. Fortunately, these levels are rather independent. So we shall present them in sequence

1. The coarse level is how does the algorithm select every possible pair of origin and destination **block-node**'s.

2. The intermediate level is how, given an origin and a destination **block-node**'s, the algorithm selects efficiently every pair of origin and destination **op-ref:atomic:rw**, that are likely to have a dependence.

3. The finest level is how to detect that two memory references are effectively in dependence. At this level appear the classical algorithms known as gcd test, separability test, simplex method, *etc.*

Before this, we must present the basic data structures that will be created, and the initializations to be performed.

7.1 Distances

There is a classical shape for data dependencies. A data dependence contains two parts. First is the type (*direct*, *anti*, *output*, *control*, or *value*), second is the distance. The distance is the distance in the iteration space between the origin and the destination. For each enclosing loop, it says how many iterations may have elapsed between origin and destination.

Thus the distance associates, to each enclosing loop of both origin and destination, an interval of integers inside which the number of iterations may be. Additionnaly, we store the number of loop levels enclosing the origin, that do not enclose the destination, and reciprocally.

There may be negative integers inside a distance, typically when origin and destination are in two nested loops, and the distance for the outer loop is strictly positive. Then, if the origin is at inner iteration n, and the destination is at inner iteration $n-3$, then the distance for the inner loop is -3

During the coarse level, the value propagated from the origin **block-node** to the destination contains the candidate iteration distance between them. Then the finest level analysis refines this distance, reducing it to a smaller set of possibilities (eventually empty), and this reduced distance will be used in data dependencies between these two **block-node**'s.

7.2 Masks

Each time we select a pair of **op-ref**'s, we need an important information, that is the overwritten variables between the two. This has two important uses:

- First use is to detect variables whose value has certainly not changed in the meantime. We are then allowed to simplify these variables out of equations, leading to more accurate data dependence analysis. This is more precise that a simple loop constants detection, since a variable may keep its value between two points, while being overwritten somewhere else in the loop.

- Second use is to detect variables (scalar or arrays thanks to **killed arrays analysis**) whose complete value has been overwritten. In that case, there must not be any dependencies between two references of such a variable.

This information, called a *mask*, is stored as a vector, associating each zone to one of the three status: *untouched*, *touched* (but not overwritten), and *overwritten*. Actually, there is also a *mask* for variables that are not known in the current unit, but that exist and communicate through **commons** between called subroutines. (The same thing occured for **in-out analysis**). We shall not describe this in more detail.

Therefore, the coarse level propagates a second information, which is the *mask* between the origin and destination **block-node**'s. Additionally, each **op-ref** receives at initialization its *mask* from the beginning of its **block-node**, and another one till the end of its **block-node**. Other interesting *mask*'s are computed, such as the *mask* from any **block-node** to the exit of its enclosing loop, to any **block-node** from the entry into its enclosing loop, and through one loop.

7.3 The two-level *cfg* traversal

First thing is to choose each **block-node** in turn, to be the initial one for data dependencies. This is done by a top-down traversal of the tree of nested loops, with each level traversed in *dfst* order.

For each selected **block-node**, we build an initial *distance*, and an initial, empty *mask*. Then we propagate the *distance* and the *mask*, following the *cfg*, towards all reachable **block-node**. The *distance* is updated each time we cross a loop boundary. The *mask* is updated each time it traverses a **block-node**. The way *mask*'s are merged when the control flow merges is natural.

For each **block-node** reached, we gather all the incoming *masks* into a single one, take the incoming *distance*, and call for the **block-node** to **block-node** analysis with these parameters.

There are some parameters that modify the way this traversal is done. It is possible to traverse each loop twice, therefore treating the first iteration separately. This may cost a lot of execution time, but eventually gives better results (fewer dependencies) It is also possible to neglect all dependence analysis outside of loops. While this saves time, the cost is that we loose read-write navigation inside the unit, and we have less dead code detected (code dead because not used).

7.4 The block-node to block-node analysis

The idea is to compare each **op-ref** from the origin **block-node** with each one from the destination. This is very expensive if we have no way to reject quickly the pairs of **op-ref**'s that are clearly not in dependence. This is where the big Zones appear (*cf* section 2).

Suppose we have n **op-ref**'s in each **block-node**. Then the cost in time is approximately $O(n^2)$. Suppose now that these op-ref's are evenly distributed into Z Zones. By definition of Zones, we are sure that if two **op-ref**'s are in different Zones (in their **varinfo** field), then they may never overlap. Then for each Zone, the time cost is $O(n^2/Z^2)$, and the time cost for the whole jobs is $O(n^2/Z)$

So the algorithm is schematically:

```
For each big Zone Z,
 OL = List of op-refs in Origin, for Z (precomputed)
 DL = List of op-refs in Destination, for Z (precomputed)
 For each OP1 in OL
  Mask1 = Mask . mask from OP1 to exit of Origin (precomputed)
  For each OP2 in DL
   Unless OP1 is a read and OP2 is a read
   Mask2 = Mask1 . mask from the start of Destination to OP2 (precomputed)
   When zones(OP1), zones(OP2) and Mask2 intersect
    analyse dependencies between OP1 and OP2
```

7.5 The op-ref to op-ref dependence analysis

The only case worth discussion is the dependence between two arrays. The classical method leads to solving an equation between two expressions, each coming from the index expressions of the two arrays. If the expressions may be proved never equal, then there is no dependence. Else, there must be a dependence in the *dfg*.

There are many tactics available. We chose some of them, and performed them in the following order:

- **GCD** test. The analysis may stop here, with certainly no dependence.
- **Separability** tests (weak and strong). The analysis may stop here, with certainly no dependence or certainly a dependence.
- **Simplex** test. This is the catch-all case. It works seldom, but may prove that there is no dependence.

- In all other cases, we must assume there is a dependence

Moreover, when the Separability test works, it is able to restrict the *distance*, giving more precision to the result.

All these tests work much better when the bounds of the variables are known with good precision. This is the justification for **variable bounds** analysis.

Also, it is good to have a variable replaced by a linear function of the loop counters, and this is why we needed to detect **induction variables**.

Lastly, `injective` arrays are useful, since when the equation is $T(exp_1) = T(exp_2)$, and T is injective, then the equation is equivalent to $exp_1 = exp_2$, which may then eventually be proved false, removing one dependence.

7.6 Additional loop control dependencies

So far, there were no control dependencies set from the **op-ref:atomic:loop-counter** to any **op-ref:atomic** inside the loop. As we said before (**context of control** analysis), this is to keep the possibility to detect **loop invariant code**.

We are now ready to put just the necessary control dependencies. At the end of **data dependence** analysis, we have now direct dependencies from the write of the loop index to each use of it in the loop. This will be just as good as a control dependence. There is still one case where code that does not use the loop index should depend on the **op-ref:atomic:loop-counter**. This is when a value set at one iteration is used at next iteration, just like in

```
do i=1,100
  b = a + 1
  a = f(b)
enddo
```

Now that we have the data dependencies, we can detect this situation. We then add a control arrow (of distance 0), from the loop counter to every **op-ref:atomic:rw:read** that has an incoming direct dependence of distance 1 or more, with respect to the loop. It is easy to check on the above example that this would set a dependence towards the read of `a`, and therefore , the two instructions have a chain of dependence from the loop counter.

Figure 9 illustrates the result of the above analyses for the small program of figure 8. We show the resulting *dfg* after all dependencies are set. We remind the relation between the *dfg* and the *cfg* by showing the underlying

nest structure. For clarity, we only show dependence distances different from 0. We don't show the anti dependences towards the writes of k and i, since they will soon be removed because these are induction variables.

```
      subroutine EX(A,B,C,Test)
      integer A,B,C,Test
      dimension A(-10:100), B(300),C(300),Test(100)
      integer i,k
C
      k = 0
      do i=1,100
        k = k + 2
        if (Test(i).lt.0) then
          B(k) = 10
        endif
        A(i) = A(i-5) + B(k)*C(k)
      enddo
      print *, k
      end
```

Figure 8: *A small unit*

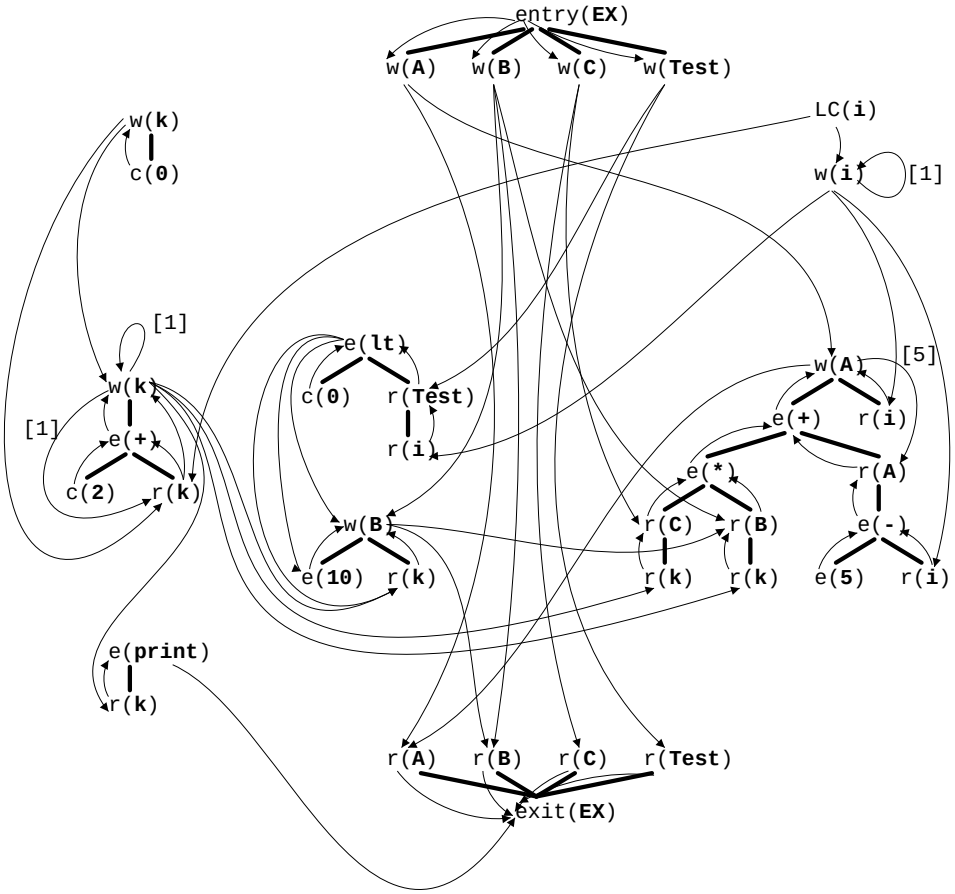

Figure 9: Final state of the *dfg*