CS-Note[*] 43

Bill J. Ellis
April 11, 2003

Refining Constant Path Analysis (CPA)

Contents

[*]Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

1 Introduction

In CS-note 34 a new technique for extracting information from programs is presented. In CS-note 39 this is described as *constant path analysis* (CPA). Further, an algorithm (ALG-CPA-1) was presented to generate *constant path descriptions* (CPDS).

In writing this algorithm and considering the application of the CPDS in more detail, some refinements have been proposed. This note explains the reasons behind the refinements and their impact on ALG-CPA-1.

2 Assumptions

ALG-CPA-1 was designed to accommodate all SPARK code. This leads to the consideration of some unusual cases which, in practise, are unlikely to occur. More importantly, if these unusual cases do occur, it is unlikely that any useful information will be found in their analysis, as the heuristics will be designed for the normal cases. Thus it is helpful to explicitly exclude these unusual cases, rather than try to artificially support them. Based on this approach some minor assumptions about the code under consideration are made.

- **Only constant paths -** In CS-note 39 it is possible to consider different paths on loop iterations by following a flow of truth values that has different sequence for different loop iterations. This flexibility is removed, focusing on the concept of constant paths.

- **The code is SPARK -** In CS-note 39 the code was being considered as a general imperative programming language. Constraints imposed by SPARK are sensible and allow for a more straight generation of the CPDS. In particular the constraints on control flow composition described in [1] §10.2 remove some complexities.

- **All loops can loop -** There always exists at least one path from a loops invariant back to the same loop invariant. Note that this path might never be logically taken. This is a minor assumption as structurally non-looping loops should always be implemented without the loop construct. This case can only occur in very poor code[1].

- **All loops can exit -** Loops will contain at least one path from a loops invariant to the loops exit. Structurally non-terminating loops may occur at the top level of well written programs. Their analysis is not being considered here.

- **All loops do exit -** Loops will always terminate. The generated CPDS and the heuristics that apply to these are all based on this assumption. Note that if SPARK were to support a proof of termination it may be possible to apply CPA inspired techniques to automate these proofs.

[1] I have written code with such loops! In shell programming it is possible and reasonable to use loops and exit commands to implement a kind of exception handling. However, I very much doubt their is any reasonable case to argue in favour of structurally non-looping loops in a strong programming language such as SPARK.

3 Generating CPDS

3.1 Record facts as test or assignment

Every item added to a CPD is a Boolean fact. These may be introduced from a test via a conditional statement or as an assignment via assignment to a variable. The facts introduced from tests or assignments may look the same. For example, in Figure 1 the conditional and assignment statements introduce the same facts. As such it is helpful to record if facts are introduced via a test or assignment within the CPDS.

	Test	**Assignment**

Code	`if (I=3) then`
	`...`
	`end if`
		`I:=3;`

Fact	$I_n = 3$	$I_n = 3$

Figure 1: Tests and assignments introducing same fact

3.2 Variable status after leaping (Avoid)

In CS-note 39 §5.13 a possible improvement to the generation of properties within CPA is discussed. The idea is to exploit variable dependencies to determine the maximum amount of information that remains available when a loop exits.

It is conjectured that the theory is valid, as it is essentially a different application of the proven information flow analysis. However, the implementation is involved and any gains are likely to be small. It might be fairly common for a variable to be unaffected within a loop and this could be detected automatically through the technique. However, the same information can often be extracted using quite light weight symbolic analysis of the CPDS[2].

[2]If variables remain unchanged then their values on entering the loop, iterating around the loop and leaving the loop will be the same. Problems can arise where unexplored subprogram calls result in variable information being lost. If using the information flow strategy the subprograms derives annotation would have revealed the details of the subprogram.

3.3 Recording structure of conditional statements

In CS-note 39 §5.13, as part of the consideration for capturing variable status after leaping it is suggested that the implementation would require recording the end of conditional statements. Although the ideas that suggested this have been dismissed above, the change to the treatment of conditional statements is valuable for other reasons.

Consider the code fragment in Figure 2. The assignment `K:=I` is structurally enclosed within an `if` statement, conditional on `Arr(I) < Arr(K)`. The following assignment `I:=Index_Type'Succ(I)` is outside this structure. Where not recording the end of conditional statements this potentially useful structural information is lost. Further, the changes required to maintain this information are quite straight forward[3].

```
...
loop
    if Arr(I) < Arr(K) then
        K := I;
    end if;
    exit when I=U;
    I:=Index_Type'Succ(I);
end loop;
...
```

Figure 2: Find smallest code fragment

3.3.1 Conditional statement: In the flowchart

The flowchart can be annotated with the full details of all conditional statements. A strategy is outlined in §4.

3.3.2 Conditional statements: In the CPDS

Every test and assignment will occur within a context of previous tests. By recording this context the CPDS will contain all of the conditional structure information. As noted earlier it is helpful to record facts introduced through tests or assignments separately. This separation can be maintained by assigning a unique number to each fact introduced through a test. These numbers can be referenced to describe the context of each test and assignment. Thus, for example, in Figure 3 if the result statement is reached the details in Figure 4 are produced.

```
if A=B then
    if I>10 then
        Result := Result + 1;
    end if;
    Count:=Count+10;
end if;
```

Figure 3: Nested conditional statement

Test
(1) $[] \to A = B$
(2) $[(1)] \to I > 10$
Assignment
$[(1), (2)] \to result = result + 1$
$[(1)] \to count = count + 10$

Figure 4: Constant path details

[3]Arguably ALG-CPA-1 did record the full details of conditional statements. The end of conditional statements can be found where their alternate branches in the flowchart re-converge. The real main change here is considering conditional statements more explicitly.

3.3.3 Dealing with exit conditionals

Nested conditional statements do not present a problem as the beginning and end of the conditional is directly identifiable. Problems arise where an `exit` command is used to jump out of a conditional statement, breaking the normal flow.

SPARK restricts `exit` commands to single nesting in single branch conditional statements inside the outermost scope of a loop. Thus the code in Figure 5 is not possible. It is conjectured that such code does not present any major difficulties to analysis. However, as working with SPARK, it is reasonable to adopt the SPARK restrictions and develop the slightly more straight forward algorithm this entails.

```
loop
    if A then
        S1
    else
        if B then
            S2
            exit;
        else
            S3
        end if;
    end if;
end loop;
```

Figure 5: Illegal SPARK code

As a result of the SPARK restrictions, only two different kinds of conditional statements will be seen, as shown in Figure 6[4]. Note that potential statement may contain nested regular conditional statements but not exit conditional statements.

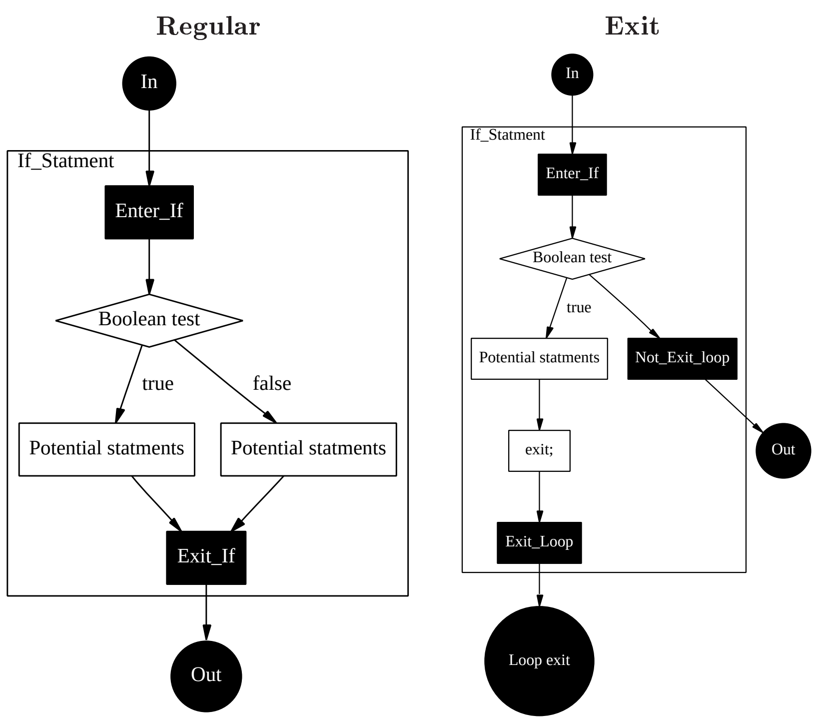

Figure 6: Two kinds of conditional statement

[4]Conditional statements with a `then` and `else` branch where both include the `exit` command is one of the possibilities excluded by SPARK.

The guard of a loop is the loops exit test. This will be true when the loop exits and false when the loop continues to iterate. Where dealing with constant paths the exact tests encountered in the loop are known. It is possible to construct the complete conditions for continuing to iterate around a loop and for exiting this loop.

Where reaching the terminals of exit conditionals different actions are taken to construct these guards:

- **Not_Exit_loop -** The test of this conditional is part of the loops not exit property. If the restriction on not allowing the nesting of **Exit** conditionals were lifted, the fact would be the negation of the conjunction of all the tests required to reach the **exit** command. For example, in Figure **??** the negated guard would be $not(A \wedge B)$.

  Note that SPARK still allows a number of exit points in a loop. Thus the loops complete exit property is only found once every reachable **Not_Exit_loop** has been reached.

- **Exit_loop -** The test of this conditional is the complete part of the loops exit property.

3.4 Flatten

Many history variables will refer to other history variables. It is possible to flatten these structures by replacing every referenced history variable with the value of the history variable. For example:

$$\begin{array}{cc} i_1 = Index'First & \wedge \\ i_2 = i_1 + 1 & \wedge \\ result_1 = 0 & \wedge \\ result_2 = result_1 + (element(data\_set, i_1)) & \wedge \\ result_3 = result_2 + (element(data\_set, i_2)) & \end{array}$$

Can be presented as:

$$\begin{array}{cc} i_1 = Index'First & \wedge \\ i_2 = Index'First + 1 & \wedge \\ result_1 = 0 & \wedge \\ result_2 = 0 + (element(data\_set, Index'First)) & \wedge \\ result_3 = (0 + (element(data\_set, Index'First))) + (element(data\_set, Index'First + 1)) & \end{array}$$

This operation is not expected to be useful where generating the initial CPDS as it loses important association information. However, for analysis, this flatten operation reveals more clearly the patterns from which properties might be generated.

3.5 Generalised facts

From examining the examples of CS-note 39 it is clear that the CPD on repeated iterations of a loop are the same, except for increasing variables history subscripts. It is advantageous to generalise these verbose traces into a smaller number of more descriptive facts.

The key to this generalisation is exploiting the association between the CPDS and loop iterations. Figure 7 shows how the CPDS are generated (solid line) from the code execution (dotted line). The repeated pattern in the CPDS for successive loop iterations is a result of encountering the same statements on each iteration. By relating the history variables of a single CPD on one iteration to the loop iteration count a single CPD can describe all iterations.

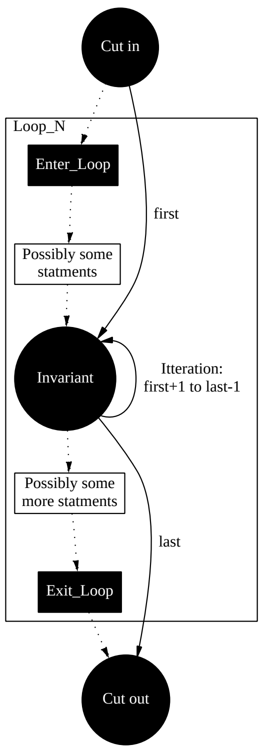

Figure 7: Properties tied to iteration

3.6 Example with counter variable

Consider an extra variable called *counter* that counts loop iterations from zero, being incremented as the invariant is crossed. This counter will be indexed by *iterator*, in range *first* to *last*, to access the value of the counter on a given iteration. Following the verbose style of CS-note 39 for a number of iterations the facts for *counter* would be:

$$\begin{aligned} counter_{succ(first)} &= counter_{first} + 1 \\ counter_{succ(succ(first))} &= counter_{succ(first)} + 1 \\ counter_{succ(succ(succ(first)))} &= counter_{succ(succ(first))} + 1 \\ &\dots \end{aligned}$$

However, it is clear that this can be reduced to a single more general statement:

$$counter_{succ(N)} = counter_N + 1$$

Which, for purely syntactic reasons, can also be expressed as the preferable:

$$counter_N = counter_{pred(N)} + 1$$

Thus the repetition seen in unwrapping M loop iterations can be reduced to a single statement in terms of N.

Note that $first$ and $last$ do not actually correspond to loop iterations (going around the loop). Thus, if a loop does not iterate $counter$ will have taken the values:

$$(counter_{(loop\_1,first)} = 0)$$
$$(counter_{(loop\_1,last)} = 1)$$

Thus even though the loop did not iterate the counter for the loop has increased from 0 to 1. The $counter$ is counting the number of times the invariant is departed and not the number of iterations. It must have some initial value and, as the invariant must be reached, the last value must be greater.

3.7 Generalised properties as an extra phase

The generalisation technique aids analysis by reducing clutter while also recording more descriptive structures. It is quite straight forward to refine ALG-CPA-1 to accommodate this change. Essentially only a single iteration of loops need be explored, indexing variables by an arbitrary iteration N rather than a concrete iteration count.

A key issue with this process is identifying when to generalise to N. Some paths through a subprogram may not encounter a loop. Further, the details wanted when trying to extract properties are more complicated than just copying the details of related paths. For example invariants must only be true when they reached. On the *last* iteration any changes to variables will never be seen by the invariant. However, any tests on the *last* iteration may reveal the value of variables on the previous iteration, which would need to be met by the invariant.

There is a need to refine the CPDS to target different kinds of property generation for different structures. This phase is an extension to the original process outlined in CS-note 39 and will be considered in more detail in a future note.

3.8 Before full generalisation

An additional phase will arrange the CPDS to target different kinds of properties. It is necessary to present the CPDS in a form suitable for the different kinds of analysis expected later. In particular it is limiting to treat invariants as different to other cutpoints.

Focus is placed on the single case of a path between cutpoints. Within this path variables can either take their value from the previous path ($pred(now)$) or from this path (now). By appropriately indexing variables with one of these classifications it is straight forward to translate to particular structures for analysis. For example, the key loop cases[5] ($first, N, pred(last)$) can be found by instantiating (now) as appropriate.

An important result of this change in representation is that the CPD of each path between cutpoints is local. It may refer to the details of a previous path using ($pred(now)$) but these details are not know here. This is quite different to the global history variable subscript seen in ALG-CPA-1.

[5]Those cases which must be met by the invariant.

3.9 Local history subscripts

Consider the code in Figure 8. From the invariant back to itself, depending on the path taken, `I` is updated either two or three times. As CPA considers constant paths, on any particular analysis the path taken will be known. If the `if` statement is entered, three assignments for `I` will be generated:

$$[] \rightarrow (I_{now} = I_{pred(now)} + 10)$$
$$[I_{now} = 7] \rightarrow (I_{now'} = I_{now} + 20)$$
$$[] \rightarrow (I_{now''} = I_{now'} + 30)$$

On each iteration it might be thought that only the final value is interesting. It would be possible to flatten these assignments and store:

$$I_{(now)} = ((I_{(loop1,pred(now))} + 10) + 20) + 30$$

However one of these assignments was a result of entering the `if` statement. The flattened expression does not record how I relates to this `if` statement in this structure. Further, `I` may have entirely overwritten in an of assignment, hiding the details of previous assignments[6]. Thus to cover the general case it remains necessary to employ history subscripts. However, these need only be local to paths between cutpoints. As a result of this all variables are sub-scripted by iteration ($pred(now)$ or now) and history ($1..N$).

```
loop
    --#true;
    I:=I+10;
    if (I=7) then
        I:=I+20;
    end if;
    I:=I+30;
    exit when I=U;
end loop;
```

Figure 8: Multiple assignments

[6] If the assignment inside the `if` statement had been: `I:=CONSTANT;` this would hide the details of the first assignment in the flattened expression.

4 Algorithm

The algorithm presented in CS-note 39 §6, ALG-CPA-1, is refined as ALG-CPA-2.

4.1 Identify all variables

The aim of this step is to identify all of the variables in the subprogram and add some properties about the status of these variables. No change is made to the behaviour of ALG-CPA-1.

4.2 Establishing the initial state

This step determines which variables can be modified by the subprogram and hence should their changes traced on paths between cutpoints. Based on these classifications the subprograms state and the initial value of this state can be determined. No change is made to the behaviour of ALG-CPA-1 at this step.

4.3 Enhance the initial state

This is a new step of the algorithm, hinted at in CS-note 39 §5.4. In some cases subprograms may be analysed within a context. In these cases it is possible that some of the arguments to the subprogram are known. If this is the case the initial state can be strengthened to include the extra information.

4.4 Construct the flowchart

This step considers building the flowchart from the source code. The behaviour of ALG-CPA-1 remains intact, being extended to include conditional statements. The flow chart is divided into a number of potentially nested sub-flowcharts. This now includes conditional statements. The two different kinds of `if` statements are treated differently:

- **If_Regular_A -** These are `if` statements of the form:
  - `if COND then ...end if;`
  - `if COND then ...else ...end if;`.

  Where there is no occurrence of the statement `exit;` in any branch of the `if` statement.

  This block of code translates to a sub-flowchart **If_Regular_A**, with a single entry point **Enter_If** and a single exit point **Exit_If**.

- **If_Exit_B -** These are `if` statements of the form:
  - `if COND then ...exit; end if;`.

  Note that due to the restrictions imposed by SPARK else branch `if` statements are ruled out here.

  This block of code translates to a sub-flowchart **If_Exit_B**. There will be a single entry point **Enter_If**. The exit point for entering the `if` statement is **Exit_If_Break_Loop** and the exit point for not entering the `if` statement is **Exit_If_Not_Break_Loop**.

  Although deducible from the flowchart, it is convenient to record which truth value of the test is required for **Exit_If_Break_Loop**.

4.5 Finding routes

Based on the assumptions in §2 the routes are now known directly. It may however be prudent to check that the assumptions hold using the searching approach of ALG-CPA-1. If the assumptions do hold, every loop must have at least one around loop and exit path. Thus the following routes exist for every loop:

- **Around loop -** From **Enter_Loop_N** to **Enter_Loop_N**.
- **Exit loop -** From **Enter_Loop_N** to **Exit_Loop_N**.

In ALG-CPA-1 these were dealt with as two separate entities. However, heuristically, some around loop paths are more naturally attached to some exit loop paths. This was described in CS-note 39 §5.7 as trying to find the closest exit paths to the around loop paths. Now that conditional statements are treated in greater detail an algorithm for for finding closest paths can be described. As a result of this it is more appropriate to think of a loop as having N pairs of (1 **Around loop**, M **Exit loop**) rather than O **Around loop** and P **Exit loop**.

As every loop must exit the non-termination path introduced in ALG-CPA-1 is removed. The only possible path through the subprogram is:

- **Termination -** From **Start** to **Stop**.

4.6 Finding paths

This step determines which paths through the code can potentially be taken. In ALG-CPA-1 this involved collecting the top level sub-flowgraphs seen on each route. The sub-flowcharts now include conditional statements and special treatment is taken to generate closer exit paths.

Note that in ALG-CPA-1 attention was given to three different ways for iterating around loops. As a result of the the more general means for describing the changes to variables between cutpoints these three distinctions have become unnecessary.

4.6.1 Establish termination path

Firstly, the single **Termination** path is found. As **If_Exit_B** conditionals must occur inside loops the **Termination** path can only contain reference to loops or regular **If_Regular_A** sub-flowgraphs. Once found, the **Termination** is recursively broken down into all of the possible flows. The step to be taken depends on the sub-flowgraph at the head of the **Termination** path:

- **If_Regular_A -** The path is copied, setting the conditional to true in the original and false in the copy. Depending on the troth value added different branches will be reached. The sub-flowgraphs for these branches are added.
- **Loop_N -** The label is left intact in the **Termination** path. The single list of sub-flowgraphs from entering the loop and back to itself are added as **Around loop** in the pair (**Around loop**, **Exit loop**). The step to be taken depends on the sub-flowgraph at the head of the **Around loop** path:
  - **If_Regular_A -** The path is copied, setting the conditional to true in the original and false in the copy. Depending on the troth value added different branches will be reached. The sub-flowgraphs for these branches are added.
  - **If_Exit_A -** These are skipped over and the next sub-flowchart is considered.
  - **Loop_N -** Leave the label intact and separately decompose the nested loop using the same strategy for this loop. Once completed the label is skipped over and the next sub-flowchart is considered.

  Once every **Around loop** has been reduced to only truth values, **If_Exit_A** and **Loop_N** the following is performed to complete each **Around loop** and add its corresponding **Exit loop**. The **Around loop** path is copied into its corresponding **Exit path**, setting every **If_Exit_A** to not exit for the **Around loop** path.

  The exit paths are then addressed. Their will be an exit path for every **If_Exit_A**. Each **If_Exit_A** is set to exit while the others are not, deleting those that occur after the exit **If_Exit_A**.

4.7 Finding flows

As a result of the more sophisticated finding paths and assumptions the flows can be added quite directly. Every occurrence of a **Loop_N** in the **Termination** path can be replaced by every combination of every (**Around loop**, **Exit loop**) pair.

Note that their is the potential for this to result in many flows to be explored. However, this is rarely the case due to the small number of condition statements and the additional pruning made by the heuristic selection of closest exit paths, rather than considering all possible exit paths.

4.8 Generating patterns from flows

The same general principals of ALG-CPA-1 are repeated, with a different style to generate the more general patterns.

4.8.1 Indexing variables

All modifiable variables are indexed using as $variable_{(iteration,history)}$. Where:

- **History -** Values of *history* are from 1 to N. Note that *history* variables are local to the path between cutpoints. Further, note that *top* is used as a convenient synonym for the last *history* variable index.
- **Iteration -** The iteration index can take two values. If set to $(pred(now))$ it refers to the previous path between cutpoints. If set to (now) it refers to this path between cutpoints. In the non loop cases $(pred(now))$ refers directly to the initial state and (now) to the single path from the start of the subprogram to the end.

Some examples:

- $i_{(now,1)}$ The first history value of variable i on this path between cutpoints.
- $i_{(now,top)}$ The last history value of variable i on this path between cutpoints.
- $i_{(pred(now),top)}$ The last history value of variable i on the previous path between cutpoints.

4.8.2 Traversing between cutpoints

Using the constant path provided from finding the flows the same approach of ALG-CPA-1 is repeated. The changes are to the treatment of conditions. As each conditional is reached it is added, recording the kind of conditional.

- **If_Regular -** A **Test stack** is used to record all of the **If_Regular_A** conditionals currently in scope. As each **Enter_If** is encountered its condition is added as a fact, either positively or its negation depending on the next truth value on the constant path. This condition will have its context added as the entire contents of the **Test stack**. The condition will be given a new unique number. The conditions number is pushed onto the top of the stack. As each **Exit_If** is encountered the top of the stack is popped off.
- **If_Exit -**
  - **Not break -** The condition is added as a fact, either positively or its negation depending on the next truth value on the constant path. The fact is added to a collection of fact held with each loop for **Not Exit**.
  - **Do break -** The condition is added as a fact, either positively or its negation depending on the next truth value on the constant path. The fact is added to a collection of fact held with each loop for **Not Exit**. The fact is added as the single member of the fact held with each loop for **Exit**.

5 Phil Thornley example

5.1 Identify all variables

data_set:	**Import**	(**Read only**)
result:	**Import**	(**write only**)
i:	**Local**	

5.2 Establishing the initial state

$$
\begin{aligned}
result_0 &= \mathbf{Undefined} \quad \wedge \\
i_0 &= \mathbf{Undefined}
\end{aligned}
$$

5.3 Construct the flowchart

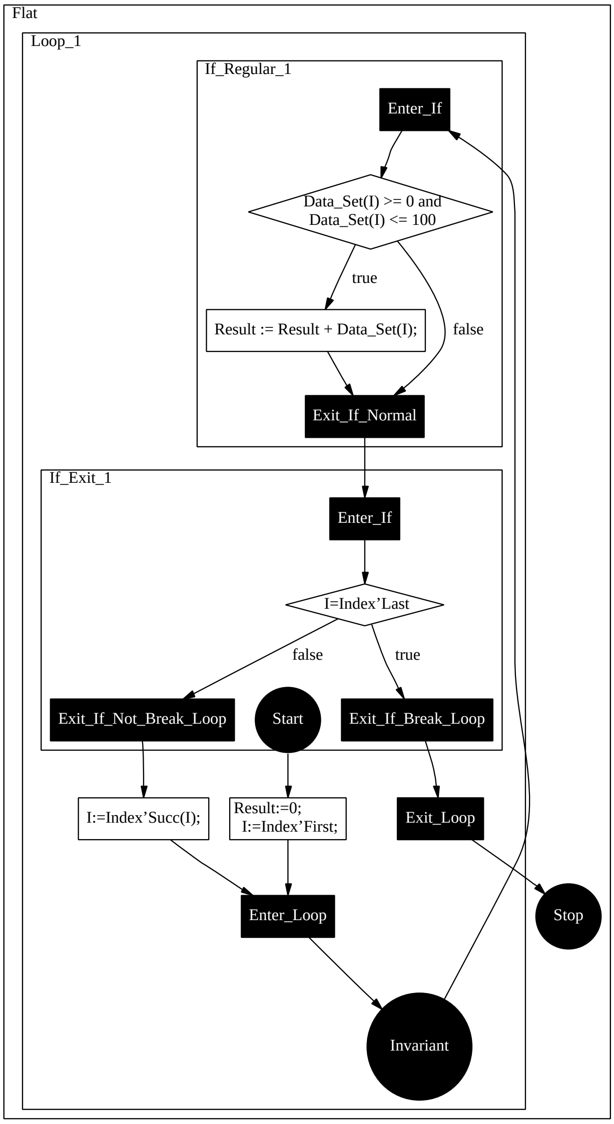

Figure 9: Flowchart

5.4 Finding paths

5.4.1 Termination

$$\begin{array}{l}\textbf{Start} \Rightarrow \textbf{Stop}\\ \quad [\textbf{Loop\_1}]\end{array}$$

5.4.2 Loop_1

$$\begin{array}{c}
(\textbf{Around loop}, \textbf{Exit loop})\\
(\textbf{Loop_1_Enter} \Rightarrow \textbf{Loop_1_Enter}, \textbf{Exit loop})\\
([\textbf{If_Regular_1}, \textbf{If_Exit_1}], \textbf{Exit loop})\\
([true, \textbf{If_Exit_1}], \textbf{Exit loop}), ([false, \textbf{If_Exit_1}], \textbf{Exit loop})\\
([true, \textbf{If_Exit_1}], [true, \textbf{If_Exit_1}]), ([false, \textbf{If_Exit_1}], [false, \textbf{If_Exit_1}])\\
([true, false], [true, \textbf{If_Exit_1}]), ([false, false], [false, \textbf{If_Exit_1}])\\
([true, false], [true, true]), ([false, false], [false, true])
\end{array}$$

5.5 Finding flows

5.5.1 Flow 1

$$\begin{array}{c}\textbf{Start} \Rightarrow \textbf{Stop}\\ [\textbf{Loop\_1}]\ [true, false, true, true]\end{array}$$

5.5.2 Flow 2

$$\begin{array}{c}\textbf{Start} \Rightarrow \textbf{Stop}\\ [\textbf{Loop\_1}]\ [false, false, false, true]\end{array}$$

5.6 Flow 1

Facts	**Assign** **Assign**	$[] \rightarrow result_{(now,1)} = \textbf{Undefined}$ $[] \rightarrow i_{(now,1)} = \textbf{Undefined}$
Facts saved as	**Initial state ⇒ Start**	

Depart cutpoint	**Start**
Path	$[true, false, true, true]$
Test stack	[]
Code	Result:=0; I:=Index'First;
Facts	**Assign** $[] \rightarrow result_{(now,1)} = Index'First$ **Assign** $[] \rightarrow i_{(now,1)} = 0$
Arriving cutpoint	**Loop_1_Invariant**
Facts saved as	**Start ⇒ Loop_1_Invariant**

Depart cutpoint	**Loop_1_Invariant**
Path	$[true, false, true, true]$
Test stack	[]
If_Regular_1	Data_Set(I) >= 0 and Data_Set(I) <= 100
Pop Path	**true** $\Leftarrow [false, true, true]$
Add facts	**Test**(1) $[] \rightarrow (element(data\_set, i_{(pred(now),top)}) \geq 0) \wedge$ $(element(data\_set, i_{(pred(now),top)}) \leq 100)$
Push test	$(1) \Rightarrow [(1)]$
Code	Result := Result + Data_Set(I);
Add facts	**Assign** $[(1)] \rightarrow result_{(now,1)} = result_{(pred(now),top)} +$ $(element(data\_set, i_{(pred(now),top)}))$
Pop test	$(1) \Leftarrow []$
If_Exit_1	I=Index'Last
Pop Path	**false** $\Leftarrow [true, true]$
Loop_1 not exit	**not exit** $[] \rightarrow not(i_{(pred(now),top)} = Index'last)$
Code	I:=Index'Succ(I);
Add facts	**Assign** $[] \rightarrow i_{(now,1)} = i_{(pred(now),top)} + 1$
Arriving cutpoint	**Loop_1_Invariant**
Facts saved as	**Loop_1_Invariant ⇒ Loop_1_Invariant**

Depart cutpoint	**Loop_1_Invariant**
Path	$[true, true]$
Test stack	[]
If_Regular_1	Data_Set(I) >= 0 and Data_Set(I) <= 100
Pop Path	**true** $\Leftarrow [true]$
Add facts	**Test**(2) $[] \rightarrow (element(data\_set, i_{(pred(now),top)}) \geq 0) \wedge$ $(element(data\_set, i_{(pred(now),top)}) \leq 100)$
Push test	$(2) \Rightarrow [(2)]$
Code	Result := Result + Data_Set(I);
Add facts	**Assign** $[(2)] \rightarrow result_{(now,1)} = result_{(pred(now),top)} +$ $(element(data\_set, i_{(pred(now),top)}))$
Pop test	$(2) \Leftarrow []$
If_Exit_1	I=Index'Last
Pop Path	**true** $\Leftarrow []$
Loop_1 exit	**Exit** $[] \rightarrow i_{(pred(now),top)} = Index'last$
Arriving cutpoint	**Stop**
Facts saved as	**Loop_1_Invariant ⇒ Stop**

6 Conclusion

In CS-note 39 the ideas of CS-note 34 were expanded and described in an algorithm. Following the completion of this note some detailed sketches of the applications for the algorithm were considered. It became apparent that the output of the algorithm was unnecessarily verbose, did not capture explicitly the patterns that arise when considering constant path analysis and lost some key information about the details of conditional statements. This note addresses these concerns by making minor but important changes to the original algorithm.

References

[1] J. G. P. Barnes. *High Integrity Ada The SPARK Approach.* Addison-Wesley, 1997.