CS-Note* 46

Bill J. Ellis

June 13, 2003

Sketches Of 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

constant path analysis CPA emerged from the ideas presented in CS-note 34. Initial emphasis was placed on developing an algorithm (ALG-CPA-1 and subsequently ALG-CPA-2) to generate *constant path descriptions* (CPDS) as described in CS-note 39 and CS-note 43. During this time various sketches of the behaviour a system taking CPDS as input were constructed. For the purpose of this note this nonexistent theoretical system will be called OLD-CPA.

As it transpired, their was a gradual mushrooming of the reasoning being proposed in OLD-CPA. The program analysis framework of Abstract Interpretation [1] was discovered as a means to control the increasingly complex process. As a result OLD-CPA became superseded before it even left the drawing board. Nevertheless, it is helpful to record the ideas surrounding OLD-CPA for future reference. Note that as OLD-CPA was never completed many of the sketches presented here are incomplete and unpolished.

2 Observations about CPA

2.1 Targets extreme values

CPA generates a separate CPD for each constant path through the program. In particular it assumes that the same path is taken through a loop on each iteration. This approach naturally targets the extreme cases of a program. This is a fundamental observation underpinning much of the analysis of OLD-CPA. The main weakness of CPA is that the constant paths it explores may not be executable in practise. This situation is considered not that significant in practise as explored in CS-note 39.

2.2 Mirroring SPARK

Each CPD is described as a collection of paths between cutpoints. SPARK generates a VC for every path between cutpoints. These paths are always the same. This similarity is by accident rather than design. The refinement of ALG-CPA-1 into ALG-CPA-2 introduced a more general notation and a tighter specification of constant paths. The result being that the points targeted by CPA correspond to the similarly minimal points of SPARK. This raised the possibility that the SPARK VCs contain sufficient information to perform the analysis of OLD-CPA, is pursued in CS-note 45. Although, theoretically, it would be possible to achieve some of the behaviour of OLD-CPA using only VCs as input, in practise such a process would be difficult and unintuitive.

2.3 Random distributed path analysis

An alternative to CPA is an analysis which randomly distributes the paths taken. The best that could be said from these specific cases is that this case exists. However, by generating many of these traces it will be possible to generalise to the general case. Alternatively the many cases could be compared to find the average case.

Such an approach would effectively mean running the program a number of times to generate a number of finite execution traces. Concrete inputs would not be required as the program analysis would still occur at an abstract symbolic level. The changing factor on each trace is the choice points made as each conditional statement is encountered. the leaping approach of CS-note 39 could be employed to ensure finite traces of loops.

It is difficult to envisage this approach buying anything that CPA does not already provide. To correctly deal with impossible paths it would still be necessary to reason about the condition at conditional statements. Further, the generalisation phase is likely to be non trivial.

2.4 Constant distributed path analysis

Another alternative to CPA is constantly distributed paths. For example, entering a conditional statement on every odd iteration and not on every even iteration. By tracing constant variations, properties can be generated to describe all iterations. There will be complexities in controlling such unusual program analysis, however the approach may be feasible within a CPA framework.

Note that this is sensibly applicable where it is known that the paths are genuinely constantly distributed, for example entering a branch that is conditional on a singularly increasing variable being even. Discovering a constantly distributed property will involve some reasoning. Where it is discovered that a condition is changing randomly, analysis using this approach will not be applicable. Although their is some merit in this technique, it is difficult to envisage a sufficient number of realistic examples to warrant its exploration.

3 Targeted properties

OLD-CPA was designed to support general property discovery. However, the emphasis was placed on discovering bounds on variables within loops.

3.1 Bounds on variables in loops

The bound of a variable A may have an exact value or a range of values:

- **Exact -** $A \geq X \wedge A \leq X$ *i.e.* $A = X$
- **Range -** $A \geq X \wedge A \leq Y \wedge X < Y$

Note that the general inequality notation can be used to describe both cases and is thus preferred for analysis.

3.1.1 Extreme end points

CPA targets extreme values, thus the first and last values found for a variable within a loop should correspond to the extreme end points of the bounds for this variable.

3.1.2 End point direction

Given the extreme end points X and Y of a variable A it is not known if $A \geq X \wedge A \leq Y$ or $A \geq Y \wedge A \leq X$. The missing information is the direction the extreme end points are changing. The possible directions are:

- **Before $<$ After $\rightarrow$ Increasing**
- **Before $\leq$ After $\rightarrow$ Increasing**
- **Before $=$ After $\rightarrow$ Constant**
- **Before $\geq$ After $\rightarrow$ Decreasing**
- **Before $>$ After $\rightarrow$ Decreasing**
- **Contradiction in above $\rightarrow$ Random**

Some guidance on direction can be found through the SPARK source. This is true for declared for loop iterators, which will be increasing for regular for loops or in reverse for reverse for loops. However, all of the remaining variables may move in a direction opposite to the loop counter:

```
Result:=0;
For I in Index loop
        Result := Result - I;
end loop;
```

The light direction information of for loops may be useful. However, it is necessary to consider the general case. For implementation reasons it may be more straight forward to rediscover the information of for loops using the general means.

To discover the direction in the general case the condition for for each direction can be posed to a proof planner. As comparing only the initial (before) and final (after) values the general iteration statements involving N can be simplified to these extreme points, removing all references to N. As eliminating N these problems can be quite straight forward. It is often possible to successfully formulate and solve these problems within a constraint solver. However, it is probably more appropriate to formulate goals for the planner, which may then invoke a constraint solver if appropriate.

3.1.3 Extracting direct bounds

If it is possible to express a variables extreme end points in terms of variables other than itself then these will directly from the bounds, ordered appropriately depending on direction. Note that partial direct bounds may be found. For example a lower bound found directly but not the upper bound.

An aside...
At the FORTEST meeting in Praxis following Andrews presentation "Increasing Automation for Exception Freedom Proofs" their was a short discussion about the discovery of the lower bound on R in the Phil Thornley Example. It was accepted that a lower bound on R is needed, however it was suggested that such easier bounds could be found by means other than generating and solving recurrence relations. This classification of direct bounds was formulated within OLD-CPA following a similar argument. There may be scope for exploiting direct bounds within our real program analysis.

3.1.4 Extracting sophisticated bounds

Where the bounds are not provided directly more sophisticated reasoning is required. The first extreme value of a variable is almost always known. This is a feature of imperative programs which transfer a known state top down. As a loop is entered the initial values are known. However, as the nature of loop iterations is not known the last extreme values are more difficult to discover. Note that it is possible for the first value to be unknown. The code may delay the first assignment to a variable to inside a loop. This a very unusual case that is trivial to deal with. A more realistic problem is a subprogram with multiple loops. The value of a variable on the entry to the second loop will be the potentially unknown value of the last value of the first loop. Further, where variables are parameters their initial values may only be known to be within a range.

Finding the last extreme values requires looking at the first value and exploring the increments from this value during the loop execution, as described below.

4 Target problem

Each CPD output from CS-note 43 takes the form of a number of facts describing the behaviour of the entire subprogram over a particular constant path. Initially OLD-CPA refines these facts to target specific property generation. Only targeting for invariants are considered here. However the same general approach could also be employed to target preconditions or postconditions.

Once the targeted facts have been constructed they are subjected to analysis to retrieve the results for this CPD. The process is repeated for every CPD and the results are generalised to gain the properties that are sufficiently wide to cover all cases.

4.1 Invariant discovery

4.1.1 Phase 1: Target the invariant

An invariant holds a property that is valid for all iterations:

$$\text{invariant properties} \left\{ \begin{array}{l} \textbf{first} \\ \textbf{succ(first) to pred(last)} \\ \textbf{pred(last)} \end{array} \right\}$$

Note that the last iteration, describing exiting the loop, is not included.

4.1.2 Phase 2: Elaborate the targeted invariant

The target invariant is elaborated by applying a number of heuristic rules, such as:

1. ▷ **Add information for pred(last) -** The details of variables on the last path are not properties that must be kept by the invariant. The last invariant property is pred(last). Variables are added for this key point.
2. ▷ **Unchanged in path -** If a variable has no trace information for a path then it is unchanged on this path. The variable is introduced and equated with its last known value, which must also be its current value.
3. ▷ **Exact replacement -** As every variable is prefixed with a subscript they are unique across all tables. Thus, it is valid to use any variable or property of the form $A = B$ to introduce B where there is already A.
4. ▷ **Match replacement -** In the **succ(first) to pred(last)** case variables are defined in terms of iteration N. Where a variable is defined in terms of a concrete iteration value, the N can be instantiated to this value and the resulting right hand side properties added.
5. ▷ **Unchanged inside loop -** In the **succ(first) to pred(last)** case a variable A is defined at iteration N as being the same variable at iteration $pred(N)$ ($A_{(Loop\_M,N,last)} = A_{(Loop\_M,pred(N),last)}$). Thus variable A must continue to have its value on the first iteration ($A_{(Loop\_M,first,last)}$) and this is added.

4.1.3 Phase 3: Generalise

The elaborated targeted invariant is used to extract bound facts on each variable. This is repeated for each CPD. The resulting facts are generalised by replacing all contradictory facts with more general facts that eliminate the contradiction.

4.2 Phil Thorley Example

The CPDS for the Phil Thorley Example as listed in CS-note 43 is considered. The example is partially complete and only considers the path entering the if statement.

4.2.1 Phase 1: Target the invariant

Inside the if

Loop_1
first
$[\ \oslash \mid \oslash\ ] \rightarrow [\ i_{(Loop\_1,first,last)} = Index'First\ ]$
$[\ \oslash \mid \oslash\ ] \rightarrow [\ result_{(Loop\_1,first,last)} = 0\ ]$
succ(first) to pred(last)
(1) $(element(data\_set, i_{(Loop\_1,pred(N),last)}) \geq 0) \wedge (element(data\_set, i_{(Loop\_1,pred(N),last)}) \leq 100)$ (2) $not(i_{(Loop\_1,pred(N),last)} = Index'Last)$ (3) $not(i_{(Loop\_1,pred(pred(N)),last)} = Index'Last)$
$[\ (1) \mid (3)\ ] \rightarrow [\ result_{(Loop\_1,N,last)} = result_{(Loop\_1,pred(N),last)} + element(data\_set, i_{(Loop\_1,pred(N),last)})\ ]$
$[\ (2) \mid \oslash\ ] \rightarrow [\ i_{(Loop\_1,N,last)} = i_{(Loop\_1,pred(N),last)} + 1\ ]$
last
(1) $(element(data\_set, i_{(Loop\_1,pred(last),last)}) \geq 0) \wedge (element(data\_set, i_{(Loop\_1,pred(last),last)}) \leq 100)$ (2) $i_{(Loop\_1,pred(N),last)} = Index'Last$

4.2.2 Phase 2: Elaborate the targeted invariant

Inside the if

Loop_1
first
$[\ \oslash \mid \oslash\ ] \rightarrow i_{(Loop\_1,first,last)} = [\ Index'First\ ]$
$[\ \oslash \mid \oslash\ ] \rightarrow [\ result_{(Loop\_1,first,last)} = 0\ ]$
succ(first) to pred(last)
(1) $(element(data\_set, i_{(Loop\_1,pred(N),last)}) \geq 0) \wedge (element(data\_set, i_{(Loop\_1,pred(N),last)}) \leq 100)$ (2) $not(i_{(Loop\_1,pred(N),last)} = Index'Last)$ (3) $not(i_{(Loop\_1,pred(pred(N)),last)} = Index'Last)$
$[\ (1) \mid (3)\ ] \rightarrow [result_{(Loop\_1,N,last)} = \ result_{(Loop\_1,pred(N),last)} + element(data\_set, i_{(Loop\_1,pred(N),last)})\ ]$
$[\ (2) \mid \oslash\ ] \rightarrow i_{(Loop\_1,N,last)} = [\ i_{(Loop\_1,pred(N),last)} + 1\ ]$
pred(last)
(1) $(element(data\_set, i_{(Loop\_1,pred(last),last)}) \geq 0) \wedge (element(data\_set, i_{(Loop\_1,pred(last),last)}) \leq 100)$ (2) $i_{(Loop\_1,pred(N),last)} = Index'Last$
$[\ \oslash \mid \oslash\ ] \rightarrow i_{(Loop\_1,pred(last),last)} = \begin{bmatrix} 1 \triangleright i_{(Loop\_1,pred(last),last)} \\ 4 \triangleright i_{(Loop\_1,pred(pred(last)),last)} + 1 \\ 3 \triangleright Index'Last \end{bmatrix}$
$[\ \oslash \mid \oslash\ ] \rightarrow result_{(Loop\_1,pred(last),last)} =$ $\begin{bmatrix} 1 \triangleright result_{(Loop\_1,pred(last),last)} \\ 4 \triangleright result_{(Loop\_1,pred(pred(last)),last)} + element(data\_set, i_{(Loop\_1,pred(pred(last)),last)}) \end{bmatrix}$

4.2.3 Phase 3: Generalise

A sketch of the final phase of this example does not exist. As it is difficult to try are recreate exactly how this should have ended, the final phase will be skipped.

4.3 Find smallest example

The find smallest example, shown below, is considered.

```
function Find_Smallest(Arr: Array_Type;
                       L, U: Index_Type) return Index_Type
is
   K: Index_Type;
begin
   K := L;
   for I in Index_Type range L+1 .. U loop
      if Arr(I) < Arr(K) then
         K := I;
      end if;
   end loop;
   return K;
end Find_Smallest;
```

4.3.1 Phase 1: Target the invariant

Inside the if

Loop_1
first
(1) $L+1 \le U$ (2) $element(Arr, i_{(Loop\_1,first,top)}) \le element(Arr, k_{(Loop\_1,first,1)})$
$[\ \oslash \mid \oslash\ ] \rightarrow k_{(Loop\_1,first,1)} = [\ L\ ]$
$[\ (1) \mid \oslash\ ] \rightarrow i_{(Loop\_1,first,top)} = [\ L+1\ ]$
$[\ (1),(2) \mid \oslash\ ] \rightarrow k_{(Loop\_1,first,top)} = [\ i_{(Loop\_1,first,top)}\ ]$
succ(first) to pred(last)
(4) $not(i_{(Loop\_1,pred(N),top)} = U)$ (5) $not(i_{(Loop\_1,pred(pred(N)),top)} = U)$ (6) $element(Arr, i_{(Loop\_1,N,top)}) \le element(Arr, k_{(Loop\_1,pred(N),1)})$
$[\ (1),(4) \mid (5)\ ] \rightarrow i_{(Loop\_1,N,top)} = [\ i_{(Loop\_1,pred(N),top)} + 1\ ]$
$[\ (1),(4),(6) \mid (5)\ ] \rightarrow k_{(Loop\_1,N,top)} = [\ i_{(Loop\_1,N,top)}\ ]$
last
(7) $i_{(Loop\_1,pred(N),top)} = U$
$[\ \oslash \mid \oslash\ ] \rightarrow function\_return_{(Loop\_1,last,top)} = [\ k_{(Loop\_1,pred(last),top)}\ ]$

Outside the if

Loop_1
first
(1) $L + 1 \leq U$ (2) $not(element(Arr, i_{(Loop\_1, first, top)}) \leq element(Arr, k_{(Loop\_1, first, 1)}))$
$[\ \emptyset \mid \emptyset\ ] \rightarrow k_{(Loop\_1, first, top)} = [\ L\ ]$
$[\ (1) \mid \emptyset\ ] \rightarrow i_{(Loop\_1, first, top)} = [\ L + 1\ ]$
succ(first) to pred(last)
(3) $not(i_{(Loop\_1, pred(N), top)} = U)$ (4) $not(i_{(Loop\_1, pred(pred(N)), top)} = U)$ (5) $not(element(Arr, i_{(Loop\_1, N, top)}) \leq element(Arr, k_{(Loop\_1, pred(N), 1)}))$
$[\ (1), (3) \mid (4)\ ] \rightarrow i_{(Loop\_1, N, top)} = [\ i_{(Loop\_1, pred(N), top)} + 1\ ]$
last
(1) $i_{(Loop\_1, pred(N), top)} = U$ (2) $not(i_{(Loop\_1, pred(pred(N)), top)} = U)$
$[\ \emptyset \mid \emptyset\ ] \rightarrow function\_return_{(Loop\_1, last, top)} = [\ k_{(Loop\_1, pred(last), top)}\ ]$

4.3.2 Phase 2: Elaborate the targeted invariant

A complete consistent example is not available. However, the following (inconsistent) example is available. This illustrates a slightly different kind of targeting focusing on individual variables.

Inside the if

first	
Vars	$$k_{(Loop\_1, first, 1)} \left\{ \begin{array}{c\|c} < & \triangleright U \\ \hline \leq & \triangleright U - 1 \\ \hline = & L \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \wedge$$ $$k_{(Loop\_1, first, last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{c} i_{(Loop\_1, first, last)} \\ \triangleright L + 1 \end{array} \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \wedge$$ $$i_{(Loop\_1, first, last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \triangleright U \\ \hline = & L + 1 \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \wedge$$
Props	$$L + 1 \leq U \quad \wedge$$ $$(element(arr, i_{(Loop\_1, first, last)}) < element(arr, k_{(Loop\_1, first, last)})) \quad \wedge$$

succ(first) to pred(last)	
Vars	$i_{(Loop\_1,N,last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & i_{(Loop\_1,pred(N),last)} + 1 \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \quad \wedge$ $k_{(Loop\_1,N,last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{c} i_{(Loop\_1,N,last)} \\ \triangleright i_{(Loop\_1,pred(N),last)} + 1 \end{array} \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \quad \wedge$
Props	$not(i_{(Loop\_1,pred(N),last)} = U) \quad \wedge$ $(element(arr, i_{(Loop\_1,N,last)}) < element(arr, k_{(Loop\_1,pred(N),last)})) \quad \wedge$

last	
Vars	$function\_return_{(Loop\_1,last,last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & K \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array} \right\} \quad \wedge$ $i_{(Loop\_1,last,last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{c} i_{(Loop\_1,pred(last),last} \\ \triangleright U \end{array} \\ \hline \geq & \triangleright L + 1 \\ \hline > & \emptyset \end{array} \right\} \quad \wedge$ $k_{(Loop\_1,last,last)} \left\{ \begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{c} k_{(Loop\_1,pred(last),last)} \\ \triangleright i_{(Loop\_1,pred(last),last)} \\ \triangleright i_{(Loop\_1,pred(pred(last)),last)} + 1 \\ \triangleright U \end{array} \\ \hline \geq & \triangleright L + 1 \\ \hline > & \emptyset \end{array} \right\} \quad \wedge$
Props	$i_{(Loop\_1,pred(last),last)} = U \quad \wedge$

Outside the if

first	
Vars	$k_{(Loop\_1,first,last)} \left\{ \begin{array}{c\|c} < & \triangleright U \\ \hline \leq & \triangleright U - 1 \\ \hline = & L \\ \hline \geq & \oslash \\ \hline > & \oslash \end{array} \right\} \wedge$ $i_{(Loop\_1,first,last)} \left\{ \begin{array}{c\|c} < & \oslash \\ \hline \leq & \triangleright U \\ \hline = & L + 1 \\ \hline \geq & \oslash \\ \hline > & \oslash \end{array} \right\} \wedge$
Props	$L + 1 \leq U \quad \wedge$ $not((element(arr, i_{(Loop\_1,first,last)}) < element(arr, k_{(Loop\_1,first,last)}))) \quad \wedge$

succ(first) to pred(last)	
Vars	$i_{(Loop\_1,N,last)} \left\{ \begin{array}{c\|c} < & \oslash \\ \hline \leq & \oslash \\ \hline = & i_{(Loop\_1,pred(N),last)} + 1 \\ \hline \geq & \oslash \\ \hline > & \oslash \end{array} \right\} \wedge$ $k_{(Loop\_1,N,last)} \left\{ \begin{array}{c\|c} < & \triangleright U \\ \hline \leq & \triangleright U - 1 \\ \hline = & \begin{array}{c} k_{(Loop\_1,pred(N),last)} \\ \triangleright k_{(Loop\_1,first,last)} \\ \triangleright L \end{array} \\ \hline \geq & \oslash \\ \hline > & \oslash \end{array} \right\} \wedge$
Props	$not(i_{(Loop\_1,pred(N),last)} = U) \quad \wedge$ $not((element(arr, i_{(Loop\_1,N,last)}) < element(arr, k_{(Loop\_1,pred(N),last)}))) \quad \wedge$

last	
Vars	$function\_return_{(Loop\_1,last,last)}\left\{\begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & K \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array}\right\} \quad \wedge$ $i_{(Loop\_1,last,last)}\left\{\begin{array}{c\|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{c} i_{(Loop\_1,pred(last),last} \\ \triangleright U \end{array} \\ \hline \geq & \triangleright L+1 \\ \hline > & \emptyset \end{array}\right\} \quad \wedge$ $k_{(Loop\_1,last,last)}\left\{\begin{array}{c\|c} < & \triangleright U \\ \hline \leq & \triangleright U-1 \\ \hline = & \begin{array}{c} k_{(Loop\_1,pred(last),last)} \\ \triangleright k_{(Loop\_1,pred(pred(last)),last)} \\ \triangleright k_{(Loop\_1,first,last)} \\ \triangleright L \end{array} \\ \hline \geq & \emptyset \\ \hline > & \emptyset \end{array}\right\} \quad \wedge$
Props	$i_{(Loop\_1,pred(last),last)} = U \quad \wedge$

4.3.3 Phase 3: Generalise

Inside the if

$$k_{(Loop\_1,last)}\left\{\begin{array}{c|c} < & \emptyset \\ \hline \leq & \emptyset \\ \hline = & \begin{array}{l|c} \textbf{first} & \begin{array}{c} i_{(Loop\_1,first,last)} \\ L+1 \end{array} \\ \hline \textbf{succ(first) to pred(last)} & \begin{array}{c} i_{(Loop\_1,N,last)} \\ i_{(Loop\_1,pred(N),last)}+1 \end{array} \\ \hline \textbf{last} & \begin{array}{c} k_{(Loop\_1,pred(last),last)} \\ i_{(Loop\_1,pred(last),last)} \\ i_{(Loop\_1,pred(pred(last)),last)}+1 \\ U \end{array} \end{array} \\ \hline \geq & \begin{array}{l|c} \textbf{first} & \emptyset \\ \hline \textbf{succ(first) to pred(last)} & \emptyset \\ \hline \textbf{last} & L+1 \end{array} \\ \hline > & \emptyset \end{array}\right\}$$

- $k_{(Loop\_1,last)} = i_{(Loop\_1,last)}$
- $k_{(Loop\_1,last)} \geq L+1$

$$
i_{(Loop\_1,last)} \left\{ \begin{array}{c|c}
< & \oslash \\ \hline
\leq & \begin{array}{l|l} \textbf{first} & U \\ \hline \textbf{succ(first) to pred(last)} & \oslash \\ \hline \textbf{last} & \oslash \end{array} \\ \hline
= & \begin{array}{l|c} \textbf{first} & L+1 \\ \hline \textbf{succ(first) to pred(last)} & i_{(Loop\_1,pred(N),last)}+1 \\ \hline \textbf{last} & \begin{array}{c} i_{(Loop\_1,pred(last),last} \\ U \end{array} \end{array} \\ \hline
\geq & \begin{array}{l|l} \textbf{first} & \oslash \\ \hline \textbf{succ(first) to pred(last)} & \oslash \\ \hline \textbf{last} & L+1 \end{array} \\ \hline
> & \oslash
\end{array} \right\}
$$

- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L+1$

Outside the if

$$
k_{(Loop\_1,last)} \left\{ \begin{array}{c|c}
< & \begin{array}{l|l} \textbf{first} & U \\ \hline \textbf{succ(first) to pred(last)} & U \\ \hline \textbf{last} & U \end{array} \\ \hline
\leq & \begin{array}{l|l} \textbf{first} & U-1 \\ \hline \textbf{succ(first) to pred(last)} & U-1 \\ \hline \textbf{last} & U-1 \end{array} \\ \hline
= & \begin{array}{l|c} \textbf{first} & L \\ \hline \textbf{succ(first) to pred(last)} & \begin{array}{c} k_{(Loop\_1,pred(N),last)} \\ \rhd k_{(Loop\_1,first,last)} \\ \rhd L \end{array} \\ \hline \textbf{last} & \begin{array}{c} k_{(Loop\_1,pred(last),last)} \\ \rhd k_{(Loop\_1,pred(pred(last)),last)} \\ \rhd k_{(Loop\_1,first,last)} \\ \rhd L \end{array} \end{array} \\ \hline
\geq & \oslash \\ \hline
> & \oslash
\end{array} \right\}
$$

- $k_{(Loop\_1,last)} = L$
- $k_{(Loop\_1,last)} < U$
- $k_{(Loop\_1,last)} \leq U-1$

$$
i_{(Loop\_1,last)} \left\{ \begin{array}{c|c}
< & \oslash \\ \hline
\leq & \begin{array}{l|l} \textbf{first} & U \\ \hline \textbf{succ(first) to pred(last)} & \oslash \\ \hline \textbf{last} & \oslash \end{array} \\ \hline
= & \begin{array}{l|c} \textbf{first} & L+1 \\ \hline \textbf{succ(first) to pred(last)} & i_{(Loop\_1,pred(N),last)}+1 \\ \hline \textbf{last} & \begin{array}{c} i_{(Loop\_1,pred(last),last} \\ U \end{array} \end{array} \\ \hline
\geq & \begin{array}{l|l} \textbf{first} & \oslash \\ \hline \textbf{succ(first) to pred(last)} & \oslash \\ \hline \textbf{last} & L+1 \end{array} \\ \hline
> & \oslash
\end{array} \right\}
$$

- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L+1$

Generalise

Path 1:

- $k_{(Loop\_1,last)} = i_{(Loop\_1,last)}$
- $k_{(Loop\_1,last)} \geq L + 1$
- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L + 1$

Path 2:

- $k_{(Loop\_1,last)} = L$
- $k_{(Loop\_1,last)} < U$
- $k_{(Loop\_1,last)} \leq U - 1$
- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L + 1$

Purge duplicates:

- $k_{(Loop\_1,last)} = i_{(Loop\_1,last)}$
- $k_{(Loop\_1,last)} \geq L + 1$
- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L + 1$
- $k_{(Loop\_1,last)} = L$
- $k_{(Loop\_1,last)} < U$
- $k_{(Loop\_1,last)} \leq U - 1$

Contradictory pair:

- $k_{(Loop\_1,last)} = i_{(Loop\_1,last)}$
- $k_{(Loop\_1,last)} = L$

Generalise to range:

- $(k_{(Loop\_1,last)} \geq i_{(Loop\_1,last)} \wedge k_{(Loop\_1,last)} \leq L) \vee$
  $(k_{(Loop\_1,last)} \geq L \wedge k_{(Loop\_1,last)} \leq i_{(Loop\_1,last)})$

Finalise to:

- $k_{(Loop\_1,last)} \geq L + 1$
- $i_{(Loop\_1,last)} \leq U$
- $i_{(Loop\_1,last)} \geq L + 1$
- $k_{(Loop\_1,last)} < U$
- $k_{(Loop\_1,last)} \leq U - 1$

5 Conclusions

The sketched system OLD-CPA is too complicated. It tries to solve a problem using heuristics in an algorithmic way. However, some of the ideas behind OLD-CPA led to the discovery and design of a more structured approach within Abstract Interpretation.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1977.