Advanced Analysis Techniques

Mike Castro Lundin
Emad Jacob Maroun

January 2018

1 General analysis

As previously defined, the aim of program analysis is to *statically* predict properties of the *dynamic* behaviour of the program. We define that the result of an analysis will be a mapping from each node $\mathbf{Q}$ in the program graph to the defined lattice $\mathbf{Z}$:

$$\text{ANALYSIS} : \mathbf{Q} \rightarrow \mathbf{Z}$$

where $\mathbf{Z}$ is defined as a subset of the powerset of $\vec{\mathbf{z}}$, where two elements are allowed in $\mathbf{Z}$ only if they are not call equivalent (which can be found in Definition 1):

$$\mathbf{Z} \subset \mathcal{P}(\ \vec{\mathbf{z}}\ )$$

such that a stack $\vec{\mathbf{z}}$ is made up of stack frames $\widehat{\mathbf{z}}$:

$$\vec{\mathbf{z}} = \widehat{\mathbf{z}} :: \vec{\mathbf{z}}\,' \mid \varepsilon$$

and $\widehat{\mathbf{z}}$ consists of a call frame $\widehat{\mathbf{q}}$ and an abstract memory frame $\widehat{\boldsymbol{\sigma}}$:

$$\widehat{\mathbf{z}} = (\widehat{\mathbf{q}}, \widehat{\boldsymbol{\sigma}})$$

An abstract call $\widehat{\mathbf{q}}$ is defined as either a pair of nodes in the program graph and a recursion label $\widehat{\mathbf{m}}$, such that the first is the node where the function was called, the second is where the function returns, and the $\widehat{\mathbf{m}}$ holds how many nodes we part of the recursion that happened at this node. Additionally $\widehat{\mathbf{q}}$ can be the special symbol '?', representing that the current frame is not associated with any call currently (could be a block as defined by the $C ::= \{D; C\}$ semantics, or a function call before the record statement).

$$\widehat{\mathbf{q}} = (\mathbf{q_i}, \mathbf{q_j}, \widehat{\mathbf{m}}) \mid ?$$
$$\widehat{\mathbf{m}} \leq \mid \vec{\mathbf{z}} \mid \ \wedge\ \widehat{\mathbf{m}} \in \mathbb{N}$$

An abstract memory frame $\widehat{\boldsymbol{\sigma}}$ is defined by the specific analysis and will generally be a mapping of a set of targets to some abstract value.

$$\widehat{\sigma} = \mathbf{target} \rightarrow \widehat{val}$$

Depending on analysis the abstract targets could for example be variables or expressions; and the abstract values could be for example $\mathcal{P}(-, 0, +)$ for sign analysis, $\mathcal{P}(true, false)$ for live variables, or $\mathcal{P}(Q_? \times Q)$ for reaching definitions.

To argue that the given lattice $\mathbf{Z}$ is finite for a finitely long program, all the above definitions in section 1 must be finite.

Starting with $\widehat{q}$, it will be finite if all three elements are finite. The node pairs will be finite as a consequence that a finite program will result in a finite amount of nodes. Element $\widehat{m}$ represents how many frames should be repeated when recursion happens, and thus is limited by the height of the stack ($\widehat{m} \leq \mid \vec{z} \mid$). Thus if the stack is finite, then so is $\widehat{m}$.

Then the lattice will be finite if $\vec{z}$ is finite, and this is discussed in section 1.3.

1.1 Operations on the Lattice

Since the lattice has been changed to have a stack structure, it is necessary to define some basic operations in the lattice, such that the worklist algorithm may know how to for example do a union/intersection when there is more than one incoming edge to a node, or compute if a node is stable by determining if the new solution is a subset/superset of an old solution.

$$\widehat{q_1} = \widehat{q_2} \begin{cases} \text{true} & \text{if } (q_i = q_a) \wedge (q_j = q_b) \wedge (\widehat{m}_1 = \widehat{m}_2) \mid \hat{q_1} = (q_i, q_j, \widehat{m}_1) \wedge \widehat{q_2} = (q_a, q_b, \widehat{m}_2) \\ \text{true} & \text{if } \widehat{q_1} = \ \widehat{q_2} = \ ? \\ \text{false} & \text{otherwise} \end{cases}$$

$$\widehat{z}_1 =_* \widehat{z}_2 \begin{cases} true & \text{if } ((q_1, q_2, \widehat{m}_1), \widehat{\sigma}_1) = \widehat{z}_1 \ \wedge \ ((q_1, q_2, \widehat{m}_2), \widehat{\sigma}_2) = \widehat{z}_2 \\ true & \text{if } (?, \widehat{\sigma}_1) = \widehat{z}_1 \ \wedge \ (?, \widehat{\sigma}_2) = \widehat{z}_2 \\ false & \text{Otherwise} \end{cases}$$

$$\widehat{z}_1 =_m \widehat{z}_2 \begin{cases} true & \text{if } ((q_1, q_2, \widehat{m}), \widehat{\sigma}_1) = \widehat{z}_1 \ \wedge \ ((q_1, q_2, \widehat{m}), \widehat{\sigma}_2) = \widehat{z}_2 \\ true & \text{if } (?, \widehat{\sigma}_1) = \widehat{z}_1 \ \wedge \ (?, \widehat{\sigma}_2) = \widehat{z}_2 \\ false & \text{Otherwise} \end{cases}$$

$$\vec{z_1} =_q \vec{z_2} \begin{cases} \text{true} & \text{if } \vec{z_1} = \vec{z_2} = \varepsilon \\ \text{true} & \text{if } (\widehat{q_1} = \widehat{q_2}) \wedge (\vec{z_1}' =_q \vec{z_2}') \mid \vec{z_1} = (\widehat{q_1}, \widehat{\sigma_1}) :: \vec{z_1}' \wedge \vec{z_2} = (\widehat{q_2}, \widehat{\sigma_2}) :: \vec{z_2}' \\ \text{false} & \text{otherwise} \end{cases}$$

Definition 1: Call equivalence for a stack and equivalence for call instances

We first define equivalences in our analysis. Two call frames are equivalent if all their sub-elements are equivalent, or if they are both '?'. We define two types of stack frame equivalence, call equivalence ($=_*$) means they have equivalent call frames, where we disregard $\widehat{m}$, while recursive equivalence ($=_m$) requires them to have the same $\widehat{m}$. Lastly, two stacks are call equivalent if all their frames are recursively equivalent. Additionally, we will note that two stacks are (regularly) equivalent if all their sub-elements are equivalent, which is not shown in the definition.

$$\widehat{\sigma}_1 \bigsqcup \widehat{\sigma}_2 = \{[x \mapsto \widehat{\sigma}_1(x) \sqcup \widehat{\sigma}_2(x)] \mid \forall x : x \in \widehat{\sigma}_1 \vee x \in \widehat{\sigma}_2\}$$

$$\widehat{z}_1 \bigsqcup \widehat{z}_2 = \begin{cases} (\widehat{q}, \widehat{\sigma}_1 \bigsqcup \widehat{\sigma}_2) & \text{if } (\widehat{q}, \widehat{\sigma}_1) = \widehat{z}_1 \ \wedge \ (\widehat{q}, \widehat{\sigma}_2) = \widehat{z}_2 \\ \bot & \text{otherwise} \end{cases}$$

$$\vec{z}_1 \bigsqcup \vec{z}_2 = \widehat{z}_1 \bigsqcup \widehat{z}_2 :: \vec{z}_1\ ' \bigsqcup \vec{z}_2\ ' \mid \vec{z}_1 = \widehat{z}_1 :: \vec{z}_1\ ' \wedge \vec{z}_2 = \widehat{z}_2 :: \vec{z}_2\ '$$

$$Z_1 \bigsqcup Z_2 = \left\{ \begin{array}{ll} z_1 \bigsqcup z_2 & \text{if } \forall z_1, z_2 : z_1 =_q z_2 \ \wedge \ z_1 \in Z_1 \ \wedge \ z_2 \in Z_2 \\ z_1 & \text{if } \exists z_1 \in Z_1 : \forall z_2 \in Z_2 : z_1 \neq_q z_2 \\ z_2 & \text{if } \exists z_2 \in Z_2 : \forall z_1 \in Z_1 : z_1 \neq_q z_2 \end{array} \right\}$$

Definition 2: Set, element and stack operations

We also define how to join the different elements of the lattice. Two abstract memory frames are joined by joining the values of any elements they share, and otherwise taking all the rest too. Two stack frames are joined by joining their abstract memory frames. This can only be done if they have the same call frame, as otherwise would not make sense. Two stacks can be joined if they are stack equivalent, in which case their stack frames are are joined. We do not allow call in-equivalent stacks to be joined, since we want to track each path through the program independently.

$$\widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 = \begin{cases} \text{true} & \text{if } \forall x \in \widehat{\sigma}_1 : x \in \widehat{\sigma}_2 \ \wedge \ \widehat{\sigma}_1(x) \sqsubseteq \widehat{\sigma}_2(x) \ \wedge \ \forall y \in \widehat{\sigma}_2 : y \in \widehat{\sigma}_1 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\widehat{z_1} \sqsubseteq \widehat{z_2} = \ \text{iff } (\widehat{q_1} = \widehat{q_2}) \wedge (\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2}) \mid \widehat{z_1} = (\widehat{q_1}, \widehat{\sigma_1}) \ \wedge \ \widehat{z_2} = (\widehat{q_2}, \widehat{\sigma_2})$$

$\vec{z_1} \sqsubseteq \vec{z_2}$ iff $(\vec{z_1} =_q \vec{z_2}) \wedge (\widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2) \wedge (\vec{z_1}\,' \sqsubseteq \vec{z_2}\,') \mid \vec{z_1} = (\widehat{q}_1, \widehat{\sigma}_1) :: \vec{z_1}\,' \ \wedge \ \vec{z_2} = (\widehat{q}_2, \widehat{\sigma}_2) :: \vec{z_2}\,'$

$Z_1 \sqsubseteq Z_2$ iff $\forall z_1 \in Z_1, \exists z_2 \in Z_2 : z_1 \sqsubseteq z_2$

Definition 3: Set, element and stack comparison

Comparing two abstract memory frames is equivalent to what we have previously seen. A frame is a subset of another frame if all of its elements are in the other frame, and the values they map to a subsets of the elements they map to in the other frame. Stack frames are compared by ensuring that they have the same call frame, and otherwise by comparing their abstract memory frames. Likewise, stack are compared by ensuring they are call equivalent, and all their frames compare in the same way. Lastly, since Z is just a powerset, comparing two of them is done as for any other powerset.

$$\vec{z}\,[q \mapsto (q_1, q_2)] = \begin{cases} \widehat{z}[q \mapsto (q_1, q_2)] :: \vec{z}\,' & \text{if } \vec{z} = \widehat{z} :: \vec{z}\,' \\ \widehat{z}[q \mapsto (q_1, q_2)] & \text{if } \vec{z} = \widehat{z} \wedge \widehat{q} \neq \varepsilon \\ \bot & \text{otherwise} \end{cases}$$

$$\widehat{z}[q \mapsto (q_1, q_2)] = \begin{cases} ((q_1, q_2, \widehat{m}), \sigma) & \text{if } \widehat{z} = ((q_x, q_y, \widehat{m}), \sigma) \\ ((q_1, q_2, 0), \sigma) & \text{if } \widehat{z} = (?, \sigma) \end{cases}$$

$$\widehat{z}[\widehat{m} \mapsto i] = ((q_x, q_y, i), \sigma) \mid \widehat{z} = ((q_x, q_y, z), \sigma)$$

$$\vec{z}[x \mapsto v] = \begin{cases} (\widehat{q}, \widehat{\sigma}) :: (\vec{z}\,'[x \mapsto v]) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \wedge x \in \text{dom}(\vec{z}\,') \wedge x \notin \text{dom}(\widehat{\sigma}) \\ (\widehat{q}, \widehat{\sigma}[x \mapsto v]) :: \vec{z}\,' & \text{otherwise} \end{cases}$$

$$\vec{z}[A \mapsto v] = \begin{cases} \widehat{\sigma} :: (\vec{\sigma}'[A \mapsto v]) & \text{if } \vec{\sigma} = \widehat{\sigma} :: \vec{\sigma}' \wedge A \in \text{dom}(\vec{\sigma}') \wedge A \notin \text{dom}(\widehat{\sigma}) \\ (\widehat{\sigma}[A \mapsto v]) :: \vec{\sigma}' & \text{otherwise} \end{cases}$$

$$\vec{z}(x) = \begin{cases} \widehat{\sigma}(x) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \wedge x \in \text{dom}(\widehat{\sigma}) \\ \vec{z}\,'(x) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \wedge x \in \text{dom}(\vec{z}\,') \wedge x \notin \text{dom}(\widehat{\sigma}) \\ \bot & \text{otherwise} \end{cases}$$

$$\vec{z}(A) = \begin{cases} \widehat{\sigma}(A) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \wedge A \in \text{dom}(\widehat{\sigma}) \\ \vec{z}\,'(A) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \wedge A \in \text{dom}(\vec{z}\,') \wedge A \notin \text{dom}(\widehat{\sigma}) \\ \bot & \text{otherwise} \end{cases}$$

Definition 4: Operations on the stack

Lastly, we define a few often-used operations on the stack: We can change the recursion label and the start and end node of the top element of the stack or a given stack frame. We can also both read and overwrite what given elements map to on the stack or a given frame.
Additionally, we define $Basic(Z)$, which splits all elements in Z into their basic components. This means that each frame is slit, such that each element of the abstract memory frame only maps to 1 value, i.e. {0}, {+} or {-} for sign analysis. Stack frames are then created such that all combinations of each frame and each element values are present. We also define the domain function, as described before, on the stack and stack frames

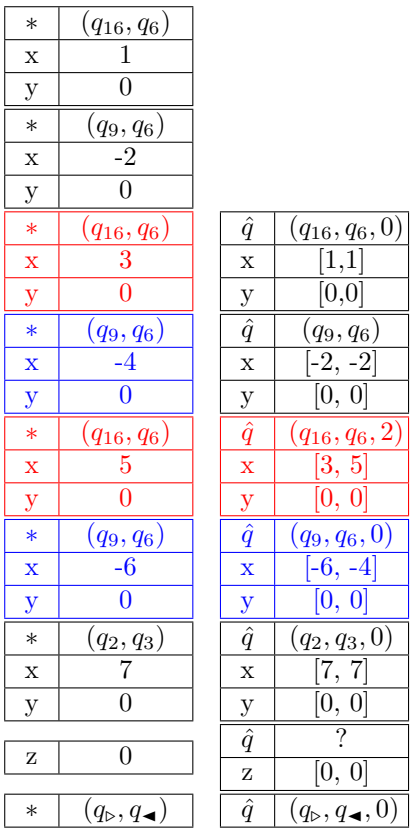
$$\text{dom}(\widehat{\sigma}) = \{x \mid \forall x : x \in \widehat{\sigma}\}$$

$$\text{dom}(\vec{z}) = \begin{cases} \text{dom}(\widehat{\sigma}) \cup \text{dom}(\vec{z}\,') & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) :: \vec{z}\,' \\ \text{dom}(\widehat{\sigma}) & \text{if } \vec{z} = (\widehat{q}, \widehat{\sigma}) \\ \{\} & \text{otherwise} \end{cases}$$

Definition 5: Domain functions

1.2 Relation between $\mathbf{Mem}_F$ and Z

The purpose of the lattice is that it will represent the state of the memory $\mathbf{Mem}_F$. In order to limit the size of the stack the record transfer function will merge when a call sequence is repeated three times in a row. This can for example be seen in Figure 1, where the red frames in $\mathbf{Mem}_F$ are merged into the red frame in $\vec{z}$, and the same happens to the blue frames.



$\mathbf{Mem}_F$ $\vec{z}$

Figure 1: $\mathbf{Mem}_F$ and its respective $\vec{z}$

This would be equivalent to using a RegEx format, and writing

$$((q_{16}, q_6, 0), \sigma[x \mapsto [1,1], y \mapsto [0,0]\ ]\ )((q_9, q_6, 0)), \sigma[x \mapsto [-2,-2], y \mapsto [0,0]\ ]\ )$$
$$[((q_{16}, q_6, 0)), \sigma[x \mapsto [3,5], y \mapsto [0,0]\ ]\ )((q_9, q_6, 0)), \sigma[x \mapsto [-6,-4], y \mapsto [0,0]\ ]\ )]+$$
$$((q_2, q_3, 0), \sigma[x \mapsto [7,7], y \mapsto [0,0]\ ]\ )(\ ?\ , \sigma[z \mapsto [0,0]\ ]\ )((q_{\triangleright}, q_{\blacktriangleleft}, 0), \sigma[\ ]\ )$$

Since repetitive calls are merged, information in regard to the amount of calls made is lost, and thus all possibilities must be considered. This means that the analysis will consider many scenarios, but the real sequence will always be under the set of possible call sequences. This can be seen in Figure 2, where a specific memory $\mathbf{Mem}_F$ is merged. This merged representation now encapsulated all possible recursion depths, as can be seen in the growing set it is equivalent to.

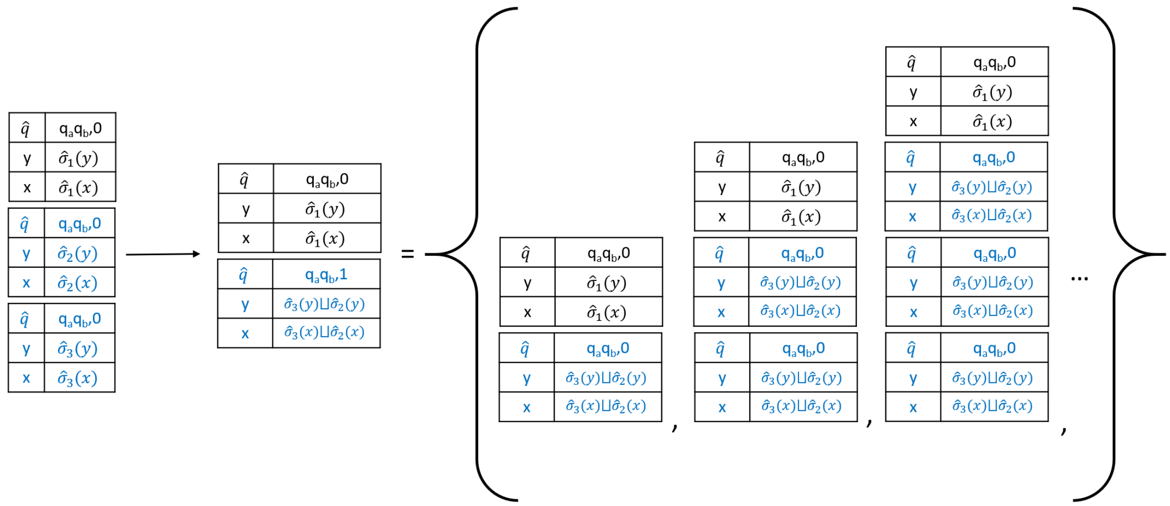


Figure 2: Merging a $\mathbf{Mem}_F$ into a $\vec{z}$ and which $\mathbf{Mem}_F$ are possible in that $\vec{z}$

1.3 Depth of the Call Stack

Since $\vec{z}$ simulates the call stack of the analysed program, we have to make sure that it has a finite length regardless of the actual length of the call stack – which could be infinite in some program executions. We do this by merging recursive calls as described previously. This only works if we assume that it is possible to merge all recursive call sequences. If this was not the case, i.e. there exists a recursive call sequence that cannot be merged, then we cannot guarantee that our analysis terminates. Therefore, we will show that for many recursive call sequences, there exists a sub-sequence that can be merged.
First, we present a few assertions:

- Infinite call stacks can only happen if recursion is used. This is easy to see, since without recursion, procedures only call other procedures. Since there is a finite number of procedures in a finite program, there must also be a finite number of calls. When we introduce recursion, infinite call stacks occur when a recursion never hits a base case.
- A recursive procedure has a finite number of places it recurses on itself. This is also easy to see, as with a finite program, the procedure declaration must also be finite, meaning there must only be a finite number of ways the procedure can call itself.

```
{ proc mutRec(x; y)
    if x=1 -> y:=1
    [] x=-1 -> y:=-1
    [] x>1 -> mutRec((-1)*(x-1); y)
    [] x<(-1) -> mutRec((-1)*(x+1); y)
    fi
  end;
  var z;
  mutRec(7; z)
}
```

Figure 3: Program for a function which recurses to distinguish if the number is even or odd

Definition: *Mutual Recursion* is usually used when a procedure calls another procedure that in turn call the first procedure. This can be extended to multiple procedures, such that for example $\texttt{P}_1$ calls $\texttt{P}_2$, which calls $\texttt{P}_3$,...,$\texttt{P}_n$ until $\texttt{P}_n$ calls $\texttt{P}_1$ restarting the mutual recursion. The grammar of the *Guarded Command Language* does not allow mutual recursion between procedures, so we repurpose the *Mutual Recursion* to describe a procedure that recurses on itself from different places. Consider the procedure `mutRec` in Figure 3. If `x > 1` on line 4, the procedure is mutually called with a negative number, meaning the next iteration will take line 5, which in turn will recurse with a positive number. Using the line numbers to identify where `mutRec` is called from, the program in Figure 3 has the following recursive call sequence before hitting the base case on line 2:

$$9 \to 4 \to 5 \to 4 \to 5 \to 4 \to 5$$

We therefore say that the recursive calls on lines 4 and 5 are *Mutually Recursive*.

Consider a recursive procedure that has 2 mutually recursive calls, which we will identify as '1' and '2'. The longest, non-merge-able recursive call sequence is 3:

$$1 \to 2 \to 1$$

If we try to add another recursive call to '1' to the above call sequence, we will have 2 calls to '1' in a row, which can therefore be merge. If we add a call to '2', we get the subsequence '$1 \to 2$' twice, which can also be merged. Doing this with '2' as the first call also yield the equivalent result. Therefore, if a procedure can recurse in two ways, we have shown that there is no way to make an infinite recursive call sequence that cannot be merged. At most we can make a call sequence of three.
This problem is equivalent to the Square-free word problem, which proves that for a vocabulary of three or more letters, it is possible to create an infinite sequence of letters with no repeating subsequence. We can therefore extrapolate that, without further restrictions, it would be possible to create an infinite call stack that cannot be merged, which would stop our analysis from terminating. With further restrictions on how calls can be made, it might be possible to inhibit such behaviour, but it is out of the scope of this chapter to show this.

2 Detection of Signs Analysis

In this chapter we will present a Detection of Signs, which extends the analysis presented in chapter 5. The analysis is still defined as a forward analysis that does over-approximation, and still makes use of the set $\mathbf{Sign} = \{-, 0, +\}$. The behaviour of evaluating arithmetic and boolean expressions is mostly preserved.
$\mathbf{sign}(\sigma) = \{x \mid \forall x \in \sigma : \mathbf{sign}(\sigma(x))\}$

2.1 Transfer Functions

Having an analysis framework that closely resembles the memory allows us to easily derive most of the transfer functions from the semantics. The transfer function for `record` must be able to handle both a normal procedure call, and recursive calls. If it is a normal procedure call, then the top of the call stack would be `?`, since we know an `enter` was executed a few transitions before. The transfer function therefore changes the top of the call stack to $(q_\bullet, q_\circ, 0)$ to reflect that this scope comes from a procedure call. In case of a recursive call, the top of the call stack would also be a `?`, but further down the stack, a $(q_\bullet, q_\circ, \widehat{m})$ would be found. To model recursion, the `record` will therefore merge stacks that it deems to be recursive. It does this by looking down the call stack, and if it sees two previous stack frames with $(q_\bullet, q_\circ, \widehat{m})$ (where it ignores the value of $\widehat{m}$), it will join these two stack frames into one, after which it converts the topmost `?` into $(q_\bullet, q_\circ, 0)$ as it normally would. This merging means the analysis does not track the whole depth of the recursive call, ensuring the analysis is finite. Keeping the topmost frame unmerged allows the analysis to recognise the base case(s) of the recursion, which could be useful in some cases. This principle is extended to merge repetitions of sequences of frames.

Like the transfer function for `record`, the one for `check` also needs to handle both normal procedure returns, and returns from recursions. It takes all states with $(q_\bullet, q_\circ, \widehat{m})$ on the top of their call stack. If $\widehat{m} = 0$ it knows that this is a normal procedure return, and just lets it though as is. If $\widehat{m} \neq 0$, it must be a recursive call, which prompts it to the following two states through instead:

1. A copy of the input state, except where $\widehat{m}$ is set to 0. This simulates the last recursive return of the call stack.

2. A copy of the input state, where the first $\widehat{m}$ call/memory frame stack have been copied (and the topmost $\widehat{m}$ set to 0) on top of the original state. This simulates the returns from each recursive call to the other.

Using these two new states, the return value can be propagated through the call stack and returned to the caller of the recursive procedure.

$$
\begin{aligned}
\widehat{S}_s[\![skip]\!]Z &= Z \\
\widehat{S}_s[\![x := a]\!]Z &= \left\{ \vec{z}[\ x \mapsto v\ ] \;\middle|\; v = \widehat{\mathcal{A}}_s[\![a]\!]\vec{z} \neq \bot \ \wedge\ \vec{z} \in Z \ \wedge\ x \in \mathrm{dom}(\vec{z}) \right\} \\
\widehat{S}_s[\![b]\!]Z &= \bigsqcup\{Z' \in Basic(Z) \mid true \in \widehat{B}_s[\![b]\!]Z'\} \\
\widehat{S}_s[\![A[a_1] := a_2]\!]Z &= \left\{ \vec{z}[A \mapsto v_2] \;\middle|\; \begin{array}{l} v_2 = \mathcal{A}_s[\![a_2]\!]\vec{z} \neq \bot \ \wedge\ A \in \mathrm{dom}(\vec{z}) \\ \wedge\ \mathcal{A}_s[\![a_1]\!]\vec{z} \cap \{+, 0\} \neq \{\} \ \wedge\ \vec{z} \in Z \end{array} \right\} \\
\widehat{S}_s[\![var\ x]\!]Z &= \left\{ \ \widehat{z}[x \mapsto 0] :: \vec{z}\,' \mid \vec{z} \in Z \ \wedge\ \vec{z} = \widehat{z} :: \vec{z}\,' \ \right\} \\
\widehat{S}_s[\![array\ A[n]]\!]Z &= \left\{ \widehat{z}[A \mapsto 0] :: \vec{z}\,' \;\middle|\; \begin{array}{l} \vec{z}) \in Z \ \wedge\ \vec{z} = \widehat{z} :: \vec{z}\,' \\ \wedge\ \mathcal{A}_s[\![n]\!]\vec{z} \neq \{-\} \end{array} \right\} \\
\widehat{S}_s[\![enter]\!]Z &= \{(?, [\ ]) :: \vec{z} \mid \vec{z} \in Z\} \\
\widehat{S}_s[\![exit]\!]Z &= \{\ \vec{z}\,' \mid \widehat{z} :: \vec{z}\,' \in Z\ \} \\
\widehat{S}_s[\![x := \lfloor a \rfloor]\!]Z &= \left\{ \vec{z}[x \mapsto v]) \;\middle|\; \begin{array}{l} v = \mathcal{A}_s[\![a]\!]\vec{z}\,' \neq \bot \ \wedge\ \vec{z} \in Z \\ \wedge\ \vec{z} = \widehat{z} :: \vec{z}\,' \ \wedge\ x \in \mathrm{dom}(\vec{z}) \end{array} \right\} \\
\widehat{S}_s[\![\lfloor x \rfloor := a]\!]Z &= \left\{ \widehat{z} :: \vec{z}\,'[\ x \mapsto v\ ]) \;\middle|\; \begin{array}{l} v = \widehat{\mathcal{A}}_s[\![a]\!]\vec{z} \neq \bot \ \wedge\ \widehat{z} :: \vec{z}\,' \in Z \\ \wedge\ x \in \mathrm{dom}(\vec{z}) \end{array} \right\} \\
\widehat{S}_s[\![record\ q_\bullet q_\circ]\!]Z &= \left\{ \begin{array}{ll} \widehat{z}_0 :: \widehat{z}_1 :: ... :: \widehat{z}_{n-1} :: \\ \quad (\widehat{z}_n \bigsqcup \widehat{z}_{2n})[\widehat{m} \mapsto n] :: \\ \qquad \widehat{z}_{n+1} \bigsqcup \widehat{z}_{2n+1} :: ... :: \widehat{z}_{2n-1} \bigsqcup \widehat{z}_{3n-1} :: \\ \qquad\quad \vec{z}\,'', & \left| \begin{array}{l} \vec{z} \in Z \\ \wedge\ \vec{z}\,' := \vec{z}\,[q \mapsto (q_\bullet, q_\circ)] \\ \wedge\ \exists n : \forall j, 1 \leq j < n : \widehat{z}_j =_q \widehat{z}_{n+j} =_q \widehat{z}_{2n+j}, \\ \quad \widehat{z}_0 =_q \widehat{z}_n \wedge \widehat{z}_0 =_* \widehat{z}_{2n} \\ \wedge\ \vec{z}\,' = \widehat{z}_0 :: \widehat{z}_1 :: ... :: \widehat{z}_{3n-1} :: \vec{z}\,'' \end{array} \right. \\ \\ \vec{z}\,[q \mapsto (q_\bullet, q_\circ)] & \left| \text{ otherwise, where } \vec{z} \in Z \right. \end{array} \right\} \\
\widehat{S}_s[\![check\ q_1 q_2]\!]Z &= \left\{ \begin{array}{ll} (\ ((q_1, q_2, 0), \widehat{\sigma}) :: \vec{z}\,'\ ) & \text{if } ((q_1, q_2, 0), \widehat{\sigma}) :: \vec{z}\,' \in Z \\ \\ \left(\begin{array}{l} \vec{z}_{copy} :: \vec{z}\,' \\ \cup \\ \vec{z}_{copy} :: \vec{z} \end{array} \right) & \begin{array}{l} \text{if } \vec{z} \in Z \ \wedge\ 0 < \widehat{m} \ \wedge \\ ((q_1, q_2, \widehat{m}), \widehat{\sigma})_1 :: \cdots :: \widehat{z}_{\widehat{m}} :: \vec{z}\,' = \vec{z} \ \wedge \\ ((q_1, q_2, 0), \sigma) :: \widehat{z}_2 :: \cdots :: \widehat{z}_{\widehat{m}} = \vec{z}_{copy} \end{array} \end{array} \right\}
\end{aligned}
$$

2.2 Extension to Interval Analysis

It is very trivial to create transfer functions for Interval Analysis from the Sign Analysis transfer functions. This will just require modifying the array transfer functions to check that at array assignment the index has an intersection with $[0, +\infty]$, and at array declaration the size is not a subset of $[-\infty, -1]$.

2.3 Correctness

To prove that our Sign analysis is correct, we have to show that given any change of the semantic state based on a semantic function, the change is reflected in the corresponding analysis state. We can specify this requirement formally as follows:
Given two program states $\vec{\sigma}_1$ and $\vec{\sigma}_2$ and analysis states z_1 and z_2 where $\beta(\vec{\sigma}_1) \sqsubseteq Z_1$ and $\beta(\vec{\sigma}_2) \sqsubseteq Z_2$ for $\beta : \widehat{State} \to \widehat{z}$ and any action α:

$$\text{if } S[\![\alpha]\!]\sigma_1 = \sigma_2 \text{ then } \widehat{S}_s[\![\alpha]\!]Z_1 \sqsubseteq Z_2$$

The β function takes a program state and returns the corresponding state in the analysis. We need to show that this holds for all actions. For brevity, we will only show that it hold for some select actions.

We start by defining β, as the whole proof hinges on it:

$$\mathbf{merge}(\vec{z}) = \begin{cases} \mathbf{merge}\left(\begin{array}{l} \vec{z}' :: \\ (\widehat{q}_{3k-1}, \widehat{\sigma}_{3k-1}) :: \cdots :: (\widehat{q}_{2k}, \widehat{\sigma}_{2k}) \\ :: (\widehat{q}_{k-1}[\widehat{m} \mapsto k], \widehat{\sigma}_{2k-1} \bigsqcup \widehat{\sigma}_{k-1}) \\ :: \cdots :: (\widehat{q}_0, \widehat{\sigma}_k \bigsqcup \widehat{\sigma}_0) :: \vec{z}'' \end{array}\right) & \begin{array}{l} \text{if } \exists\ k > 0 : \vec{z}' :: \widehat{z}_{3k-1} :: \cdots :: \widehat{z}_0 :: \vec{z}'' = \vec{z} \wedge \\ \widehat{z}_{3k-1} =_m \widehat{z}_{2k-1} =_* \widehat{z}_{k-1} \wedge \cdots \wedge\ \widehat{z}_{2k} =_m \widehat{z}_k =_m \widehat{z}_0 \wedge \\ \forall 0 \le x \le k : (\widehat{q}_x, \widehat{\sigma}_x) = \widehat{z}_x\ \wedge\ \text{for the smallest } |\vec{z}''| \text{ and } k \end{array} \\ \widehat{z} & \text{if } \widehat{z} :: \varepsilon = \vec{z} \\ \mathbf{merge}(\vec{z}') :: \widehat{z} & \text{Otherwise, where } \vec{z}' :: \widehat{z} = \vec{z} \end{cases}$$

$$\beta(\sigma) = \begin{cases} ((q_\bullet, q_\circ, 0), \mathbf{sign}(\sigma)) & \text{if } * \in \sigma\ \wedge\ \sigma(*) = (q_\bullet, q_\circ) \\ (?, \mathbf{sign}(\sigma)) & \text{Otherwise} \end{cases}$$

$$\beta(\vec{z}) = \begin{cases} merge(\beta(\sigma_n) :: \cdots :: \beta(\sigma_0)) & \sigma_n :: \cdots :: \sigma_0 = \vec{\sigma} \\ \bot & \text{Otherwise}(\vec{\sigma} \text{ is undefined}) \end{cases}$$

Case $[\![var\ x]\!]$:
For the variable declaration actions, it is easy to see that the requirement holds. if $x \in \sigma \mid \sigma :: \vec{\sigma}' = \vec{\sigma}_1$ then x is assigned to 0 in $\vec{\sigma}_2$. x would therefore also be in $\widehat{z}_1 \mid \forall \widehat{z}_1 \in Z_1$, and therefore assigned to $\{0\}$ in $\widehat{z}_2 \mid \forall \widehat{z}_2 \in Z_2$. Likewise it also holds if $x \notin \sigma$. If $\vec{\sigma}_1 = \varepsilon$ then $\widehat{z}_1 = (\varepsilon, \varepsilon)$, $\vec{\sigma}_2$ is undefined and $Z_2 = \bot = \beta(\vec{\sigma}_2)$.

Case $[\![record\ q_\bullet q_\circ]\!]$:
If the procedure call is not a recursive call, it is easy to see that the requirement holds, since $q_\bullet q_\circ$ are assigned both in the program and the analysis. We can clearly see that no merging is done by β, since without recursion, we wouldn't have $q_\bullet q_\circ$ anywhere else in the memory stack.
If the procedure call is recursive, we have three cases that we need to look at:

1. A subsequence ending in $q_\bullet q_\circ$, which is equal to the top subsequence has been called once before. In this case, no merging will happen, since we need three subsequences before merging. It is therefore easy to see that the requirement will hold, since nothing is done but assigning $q_\bullet q_\circ$.

2. A subsequence ending in $q_\bullet q_\circ$, which is equal to the top subsequence has been called twice before. $\widehat{S}_s[\![\alpha]\!]\beta(\vec{\sigma}_1)$ will merge the two previous sequences into one, setting the topmost $\widehat{m}$ to the length of the sequence. $\beta(\vec{\sigma}_2)$ will do the same, as it will see that $\vec{\sigma}_2$ has the same sequence 3 times. Therefore, this case also holds.

3. A subsequence ending in $q_\bullet q_\circ$, which is equal to the top subsequence has been called more than twice before. $\vec{z}_1$ will therefore have two of the sequences on the top (below the new one), where the second one is a merger. The transfer function will see that the new sequence is equal to them, and therefore merge the unmerge previous sequence into the merged previous sequence. $\vec{z}_2$ will see all the equivalent sequences, and merge all bu the top into one. This will therefore result in the same stack frame as the output of the transfer function, which means this case also holds.

2.4 Example

Let us consider the execution of the factorial function of Figure 4, using the program graph of Figure 5.

```
1  { proc fac(x; y)
2      if x<=1 -> y:=1
3      [] x>1 -> { var t;
4                  fac(x-1; t);
5                  y:=x*t
6                }
7      fi
8    end;
9    fac(4; z)
10 }
```

Figure 4: Example program for factorial function

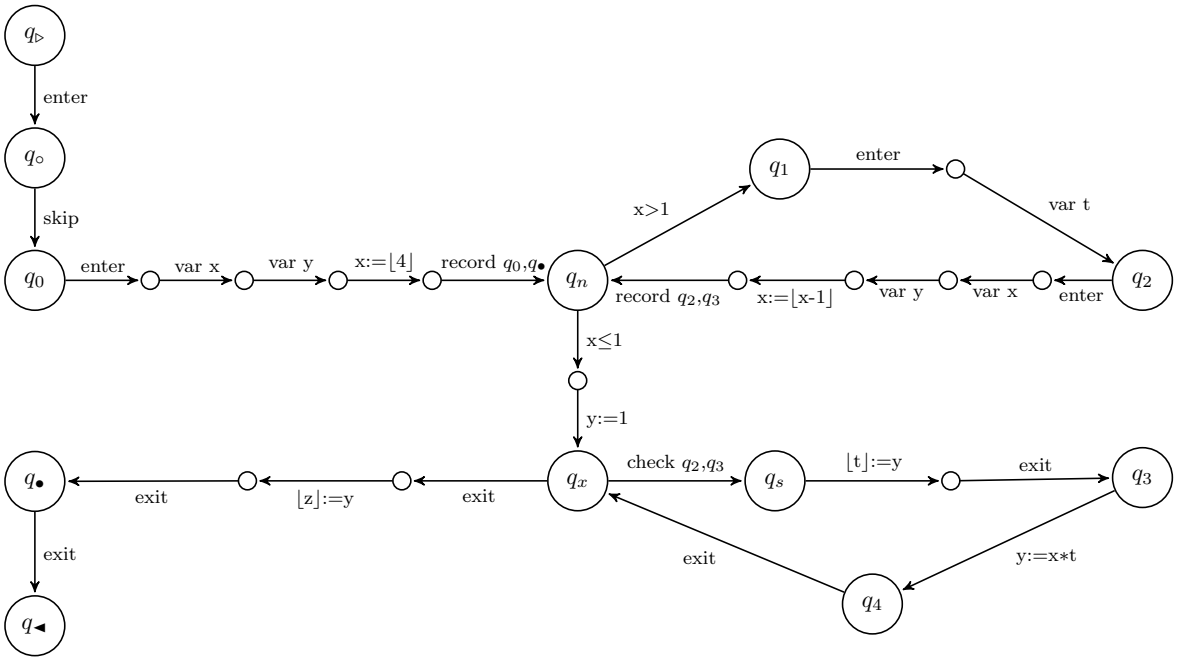


Figure 5: Program graph for factorial function of Figure 4

2.4.1 Interval Analysis

We will now compare selected memories at different program points for an execution sequence with the abstract memory arising from the analysis. In the procedure calls, the memory can be seen in Figure 6, and the abstract memory as a result of the analysis can be seen in Figure 7. It can be seen that for all points in the execution sequence except the last, the analysis at the same point will simply transform the integer into its abstract equivalent (in this case an interval); and transform the control variable * into its $\hat{q}$ equivalent. For the last point the analysis encapsulates that there are at least two instances of the repetition of the two frames, where x has a value of either 2 or 3. The case where there are two instances of the repetition, where in the top frame x has the value 2 and in the second instance it has the value 3 is one of the considered cases (as will be seen in the procedure returns). Thus the green frames in the analysis result will be an over-approximation of the memory. Only the last point of q_n in the execution sequence will go through to the procedure returns, since the others will be filtered by the boolean statement going to q_x.

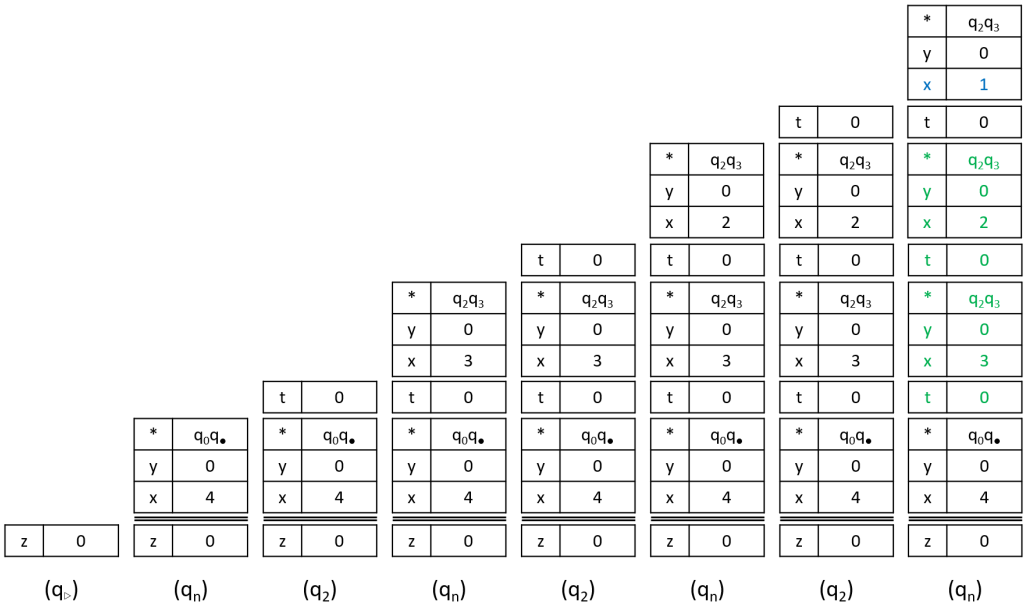


Figure 6: Procedure calls in memory

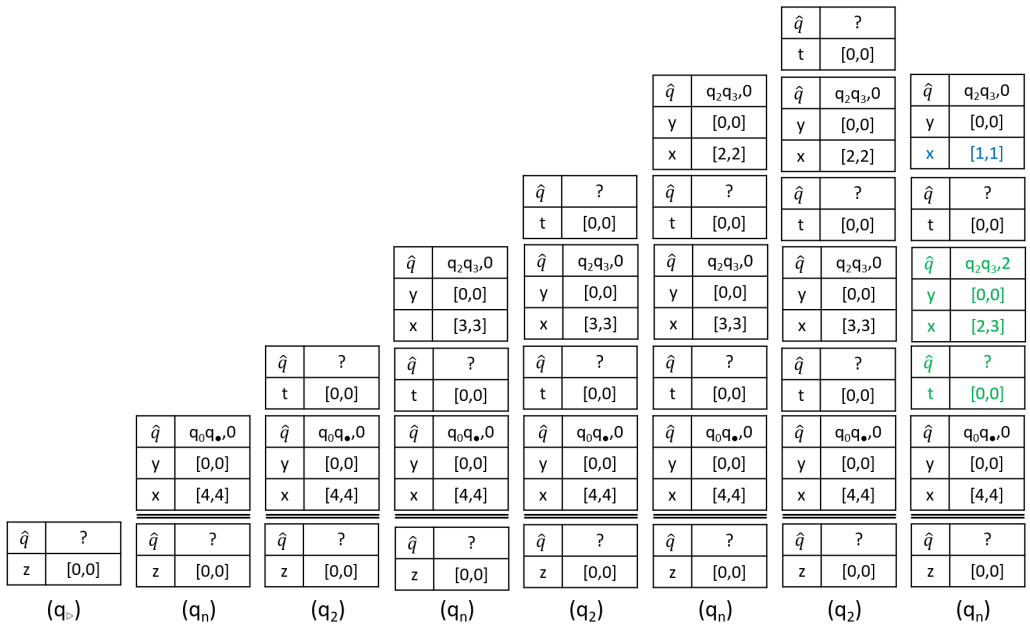


Figure 7: Procedure calls in abstract memory as a result of the analysis

For the procedure returns the memory is shown in Figure 8, while the analysis counterpart is in Figures 9, 10, 11, 12. The intervals in red in Figure 9 and Figure 11 are merged because the call stack is equivalent, leading to the interval in Figure 11. Similarly the intervals for y in the blue frames in Figure 10 and Figure 11 will get merged into the end result in Figure 12. The analysis will have to repeat the step in Figure 11 until it stabilises, which in this case will be when it reaches $+\infty$.

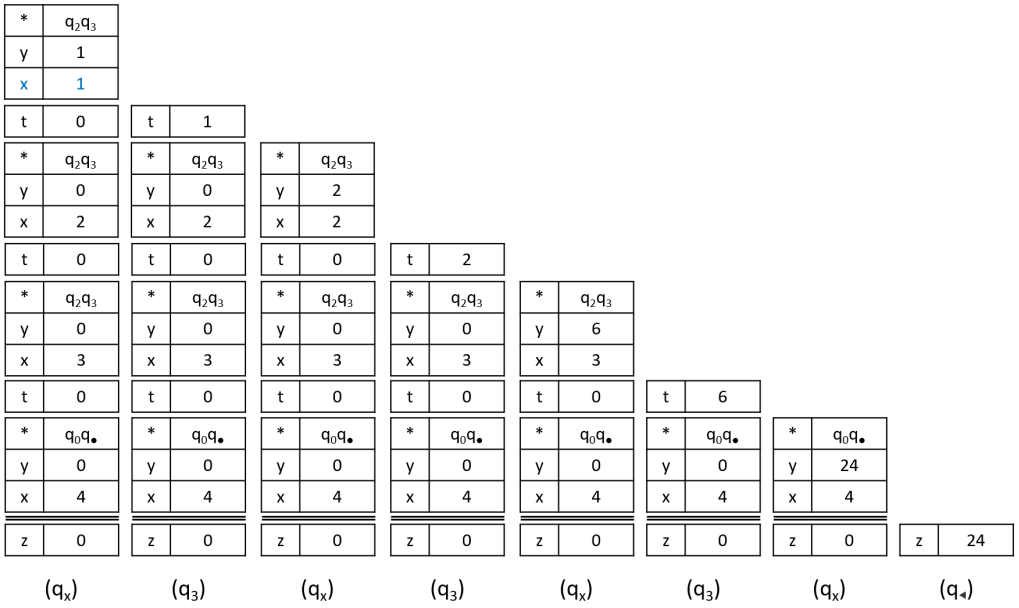


Figure 8: Procedure returns in memory

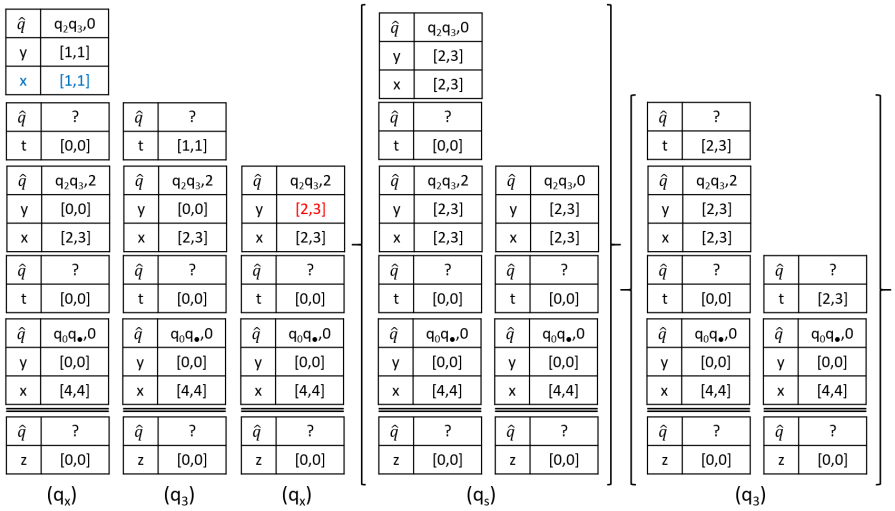


Figure 9: Procedure returns in abstract memory as a result of the analysis: Separation into two cases in q_s.

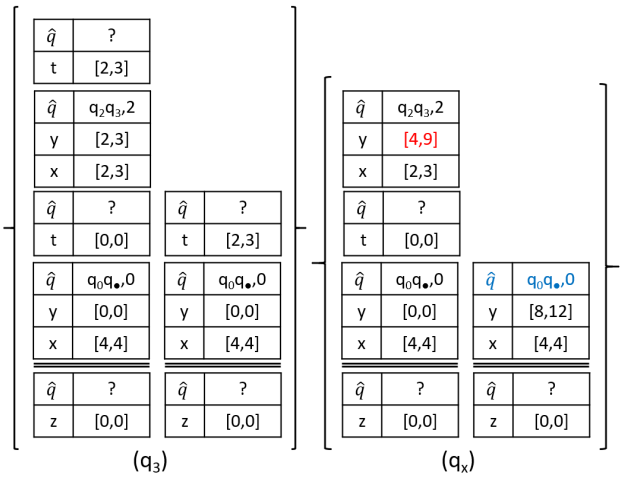


Figure 10: Procedure returns in abstract memory as a result of the analysis: First possible output set in blue

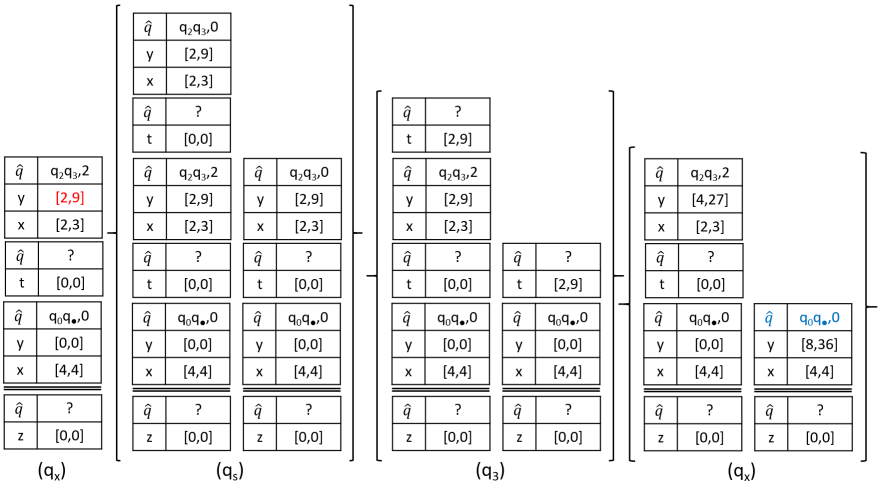


Figure 11: Procedure returns in abstract memory as a result of the analysis: Further recursion, output in blue

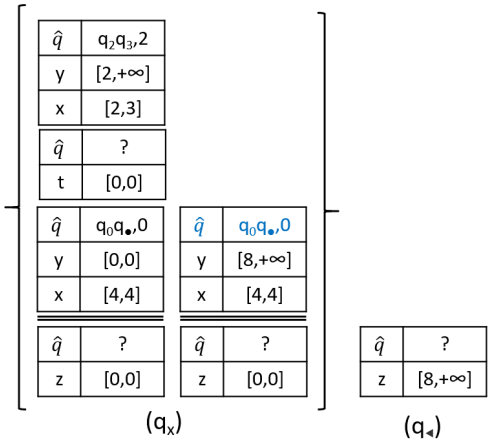


Figure 12: Procedure returns in abstract memory as a result of the analysis: End result

In order to argue that the analysis is correct for this example, then every position in the execution sequence in memory ($\mathbf{Mem}_F$) must be encapsulated by some analysis abstract state $\hat{z}$ happening in the same state. An abstract state encapsulates a position in memory if it is possible to construct a $\hat{z}'$ by expanding the labelled repetitions in $\hat{z}$ into one case in Figure 2 such that $\beta(\mathbf{Mem}_F) \sqsubseteq \hat{z}$.