Program graphs

Program Graphs: a program graph is a control flow graph with edges labelled by actions; two special nodes are identified: the initial, which represents the starting of the execution, and the final node, which represents a point where the execution will have terminate.

Program graphs are supposed to compute some output values based on some input.

A program graph consists of

- Q: a finite set of nodes
- $q_0, q_f \in Q$: the initial and the final node, respectively
- Act: a set of action
- $\rightarrow \subseteq Q \times Act \times Q$ a finite set of edges.

F. Nielson, H.R. Nielson, Formal Methods. Springer, 2019.

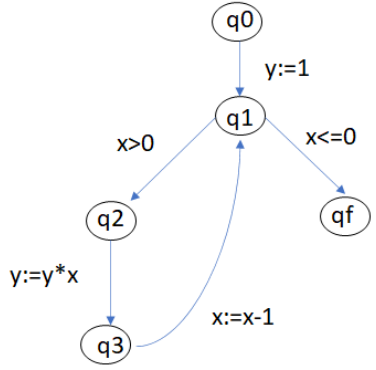
Program graph for the factorial function

Input: x Output: y function: $y = x!$

Q = $\{q_0, q_1, q_2, q_3, q_f\}$

$Act = \{y := 1, x > 0, x <= 0, x := x - 1, y := y * x\}$

$\rightarrow = \{q_0 \rightarrow_{y:=1} q_1, \cdots\}$



Program graphs

The language for the program graph above is:
$y := 1 \;\; (x > 0 \;\; y := y * x \;\; x := x - 1)^* \;\; x <= 0$

To deal with different programming languages, the language *Guarded Commands* introduced by Dijkstra in 1975 is used, and the techniques used can be extended to the more familiar languages.

Program graphs

The semantics of a program graph consists of

- the memory $m \in M$
- a semantic function specifying the meaning of the actions: given a memory before executing the action, returns the memory after the action.

A configuration is a pair (q, m), with $q \in Q$ a and $m \in M$. Memory m is assigned to state q of the graph.

The memory assigned to q_0, is named the initial memory. The factorial is computed on the value assigned to x in the initial memory.

Properties

Deterministic system

- A program graph and its semantics constitute a *deterministic* system whenever for each complete execution sequence, starting from the same initial memory, the value computed is the same.

Sufficient condition

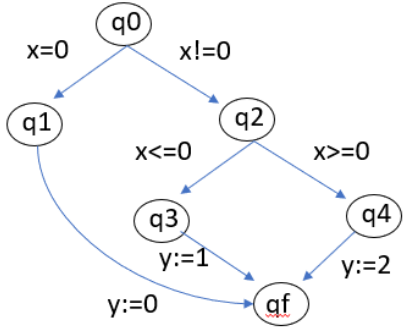
- A program graph and its semantics constitute a deterministic system whenever: distinct edges with the same source node have satisfaction conditions for the enabling of the action that do not overlap

The program graph and its semantics for the factorial function is deterministic.

Properties

The previous condition is not necessary for obtaining a deterministic system.

$x >= 0$ and $x <= 0$ overlap when x is equal to 0.



Properties

Evolving system

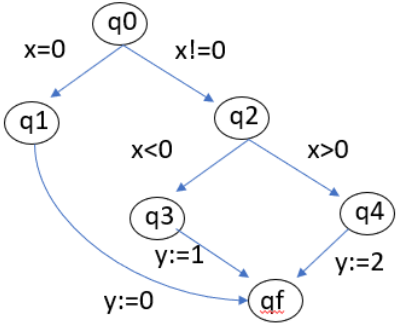
- A program does not stop in a stuck configuration (evolving system). Weaker condition that *the program always compute a value (because the program may loop)*.

- A program graph and its semantics does not stop in a stuck configuration if for every non-final node and every memory there is an edge leaving it such that the satisfaction condition for the enabling of the action is satisfied.

Properties

The previous condition is not necessary for obtaining an evolving system.

$x > 0$ and $x < 0$ have a gap exactly when x is equal to 0.



Program analysis

- A fully automated approach to prove properties.

- Express approximate behaviours of programs.
- It is a static technique, because information are obtained without the executing the program on real values.

Basic rule:

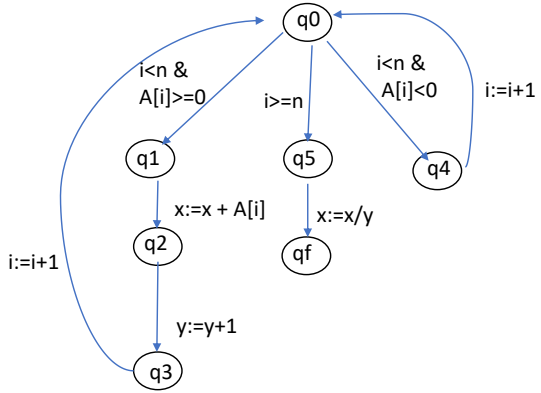
we use abstractions of the memories rather than the real memories. Generally abstractions are built by considering properties of the values.

For example:
we can abstract natural numbers to the property of being odd or even.
We can abstract integer numbers to their sign: positive, zero or negative.

An example

Computing the average of non-negative elements of an array.

Input: A[3]
Output: y = average of non-negative elements of A



initial memory m

x	0
y	0
i	0
n	3
A[0]	0
A[1]	4
A[2]	7

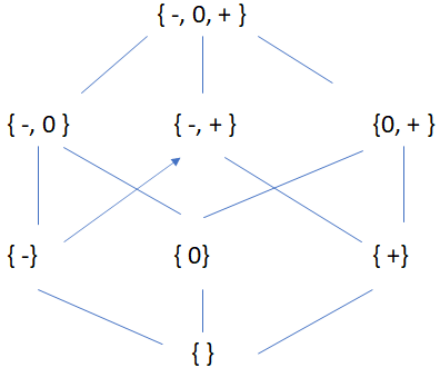
initial configuration (q0, m)

Property we want to analyse

- Division by zero

Abstract memories: values are replaced by their sign, assuming $\mathit{Sign} = \{-, 0, +\}$.

Power of *Sign*:



Examples of abstract memories

x	0
y	0
i	0
n	3
A[0]	0
A[1]	4
A[2]	7

Each element of the array A
Is abstracted into its sign.

abstract memory

x	0
y	0
i	0
n	+
A[0]	0
A[1]	+
A[2]	+

Another solution:
We can abstract the elements
of the array A into a single element
which is the union of the signs.

abstract memory

x	0
y	0
i	0
n	+
A	{0,+}

Abstract executions

Example: Addition of individual signs

$+^\circ$	-	0	+
-	{-}	{-}	{-, 0, +}
0	{-}	{0}	{+}
+	{-, 0, +}	{+}	{+}

Analysis assignment

The analysis of the program is an assignment of abstract memories with each node in the program graph

$$Assign : Q \rightarrow PowerSet(Mem)$$

where $PowerSet(Mem)$ is the set of collections of abstract memories.

Exercise: compute the assignment for the previous program graph.

Analysis assignment

The collection of abstract memories associated with a node should be an over-approximation of the set of memories that could occur at that node in an execution sequence.

If a concrete memory m occurs at node q at some point during an execution sequence then the collection of abstract memory at q must contain the corresponding abstract memory, but may contain additional abstract memories as well.