Verifying Success

We consider now methods by which we can verify the property of success, i.e. deadlock-freedom. As in the case of partial correctness, the method starts by identifying a cut-set $\mathcal{C}$ and an associated assertion network.

First, consider the case that the cut-set if full, i.e. contains all locations in the program.

Consider a node $\ell \in \mathcal{L}$ in the program. Let $c_1, \ldots, c_k$ be the guards on all edges departing from node ℓ. We define the exit condition for ℓ to be

$$E_\ell : \quad c_1 \ \vee \ \cdots \ \vee \ c_k$$

The following claim summarizes the first version of a rule for proving success.

Claim 8. *In order to prove that program P is p-successful (i.e., no p-computation ever deadlocks), it is sufficient to find a full network $\mathcal{N} : \{\varphi_\ell \mid \ell \in \mathcal{L}\}$, satisfying the following requirements:*

1. *The network $\mathcal{N}$ is inductive.*
2. $p \quad \rightarrow \quad \varphi_{\ell_0}$
3. $\varphi_\ell \quad \rightarrow \quad E_\ell$ *for every $\ell \in \mathcal{L}$*

Proof Let $\sigma : \langle \ell_0, d_0 \rangle, \ldots, \langle \ell, d_m \rangle$ be a p-computation segment reaching location ℓ. By premise 2, σ is also a φ_{ℓ_0}-computation. By Claim 1 and premise 1, $d_k \models E_\ell$. By premise 3, $d_m \models E_\ell$ which implies that at least one of the edges departing from location ℓ is enabled. Thus, σ cannot deadlock at ℓ. ∎

Extensions to Non-Full Networks

We now extend the method to apply also in the case that the network $\mathcal{N}$ is not necessarily complete. Consider a partial network $\mathcal{N} : \langle \mathcal{C}, \{\varphi_\ell \mid \ell \in \mathcal{C}\}\rangle$.

Let $\widetilde{\ell} \notin \mathcal{C}$ be a location not in $\mathcal{C}$. As previously introduced, let $\Pi_{\mathcal{C},\widetilde{\ell}}$ be the set of paths connecting a location in $\mathcal{C}$ to $\widetilde{\ell}$ without passing through any other cut-point. For each path $\pi \in \Pi_{\mathcal{C},\widetilde{\ell}}$, let $srce(\pi)$, c_π, and f_π denote, respectively, the cut-point at the beginning of path π, the summary traversal condition, and data transformation associated with π.

The following claim summarizes the general rule for proving success, using an arbitrary network.

Claim 9. *In order to prove that program P is p-successful (i.e., no p-computation ever deadlocks), it is sufficient to find a network $\mathcal{N} : \langle \mathcal{C}, \{\varphi_\ell \mid \ell \in \mathcal{C}\}$, satisfying the following requirements:*

1. *The network $\mathcal{N}$ is inductive.*
2. $p \quad \rightarrow \quad \varphi_{\ell_0}$
3. $\varphi_\ell \quad \rightarrow \quad E_\ell$ *for every* $\ell \in \mathcal{C}$
4. $\varphi_{srce(\pi)}(V) \ \wedge \ c_\pi(V) \quad \rightarrow \quad E_{\widetilde{\ell}}(f_\pi(V))$ *for every* $\widetilde{\ell} \notin \mathcal{C}$ *and path* $\pi \in \Pi_{\mathcal{C},\widetilde{\ell}}$

Proof of the Claim

Assume that there exists a network $\mathcal{N} : \langle \mathcal{C}, \{\varphi_\ell \mid \ell \in \mathcal{C}\}$ which satisfies the four requirements of Claim 9 but program P is not p-successful.

In that case, there exists a p-computation segment σ reaching some location $\widetilde{\ell}$ with data state d such that $d \not\models E_{\widetilde{\ell}}$. We consider two cases:

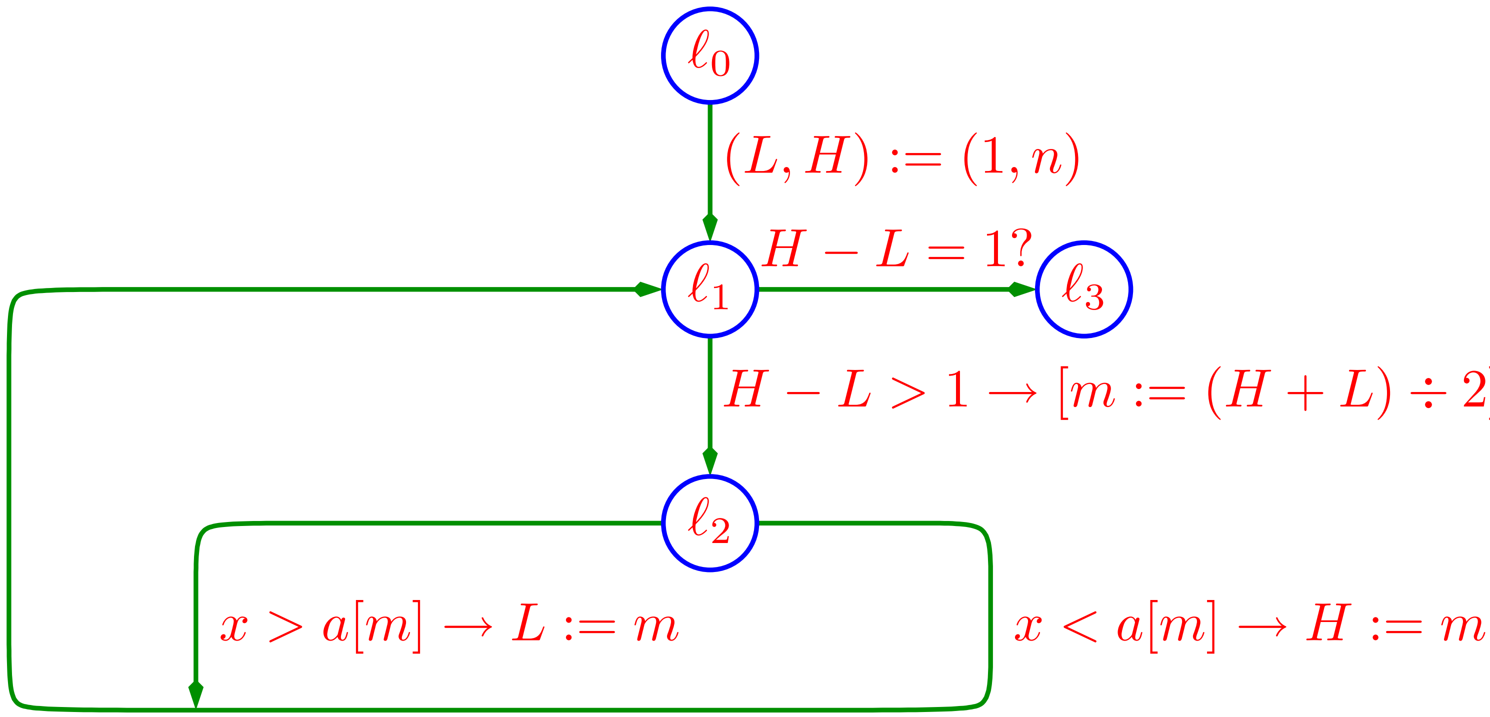
Case $\widetilde{\ell} \in \mathcal{C}$
Since $\mathcal{N}$ is inductive, data state d must satisfy $\varphi_{\widetilde{\ell}}$. However, this contradicts requirement 3 of the claim and the assumption $d \not\models E_{\widetilde{\ell}}$. Therefore, this case is impossible.

Case $\widetilde{\ell} \notin \mathcal{C}$
In this case we consider the last cut-point location visited by the execution σ. Assume that this is location ℓ, and since then σ followed the path $\pi \in \Pi_{\mathcal{C},\widetilde{\ell}}$ on its way to $\widetilde{\ell}$. Let d_0 be the data state with which σ last visited location ℓ. Since σ followed the path π, we know that $d_0 \models c_\pi$ and $d = f_\pi(d_0)$. Due to the inductiveness of $\mathcal{N}$, we also know that $d_0 \models \varphi_\ell$. Substituting these facts in requirement 4 of the claim, we conclude that $d \models E_{\widetilde{\ell}}$ which contradicts the assumption $d \not\models E_{\widetilde{\ell}}$.

We thus conclude that programs P is p-successful. ∎

Example: Binary Search

The following program is expected to identify the precise range in which an input number x falls. We are given a sorted array of numbers $a[1..n]$ such that $a[1] < x < a[n]$ and $x \neq a[i]$ for all $i \in [1..n]$.



The specification is given by $\langle p, q \rangle$, where

$$\begin{array}{ll} p: & n > 1 \ \wedge \ (a[1] < x < a[n]) \ \wedge \ \textit{sorted}(a, 1, n) \ \wedge \ \forall i : [1..n] : x \neq a[i] \\ q: & (1 \leq L < n) \ \wedge \ (a[L] < x < a[L+1]) \end{array}$$

In order to prove success for this program, we consider locations ℓ_1 and ℓ_2. Their exit conditions are respectively given by

$$E_{\ell_1} : H - L \geq 1 \qquad \text{and} \qquad E_{\ell_2} : x \neq a[m]$$

Binary Search Continued

As the cut-set, we take $\mathcal{C} = \{\ell_0, \ell_1, \ell_3\}$. The assertion network is given by

$$\begin{array}{ll} \varphi_0: & p \\ \varphi_1: & (1 \le L < H \le n) \ \wedge \ \forall i:[1..n]: x \ne a[i] \\ \varphi_3: & 1 \end{array}$$

Inductiveness of the network follows from the following property:

$$H - L > 1 \quad \rightarrow \quad L < (H+L) \div 2 < H$$

Absence of deadlock at locations ℓ_1 and ℓ_2 follows, respectively, from requirements 3 and 4 as follows:

3 for ℓ_1 : $\underbrace{\cdots \ \wedge \ L < H \ \wedge \ \cdots}_{\varphi_1} \quad \rightarrow \quad \underbrace{H - L \ge 1}_{E_{\ell_1}}$

4 for ℓ_2 : $\underbrace{(1 \le L < H \le n) \ \wedge \ \forall i:[1..n]: x \ne a[i]}_{\varphi_1} \wedge \underbrace{H - L > 1}_{c_\pi} \quad \rightarrow \quad \underbrace{x \ne a[(H+L) \div 2]}_{E_{\ell_2}(f_\pi(V))}$

Freedom from Faults

The correctness criterion and proof method for deadlock absence can be extended to guarantee absence of faults. Examples of faults during execution are: accessing a variable which has not been assigned a value, an array access with an out-of-range subscript, division by 0, arithmetic overflow, extracting the square root of a negative argument, etc.

An execution which does not generate any faults is called a fault-free execution. A program whose p-computations are all fault-free is called p-fault-free.

For an assignment $\alpha : V := f(V)$, it is possible to formulate a safety condition Γ_α which guarantees fault freedom in the execution of α. For example, the safety condition for the assignment $\alpha : A[i] := sqrt(A[j]/k)$ is given by

$$\Gamma_\alpha : \quad i \in range(A) \ \wedge \ j \in range(A) \ \wedge \ k \neq 0 \ \wedge \ A[j]/k \geq 0$$

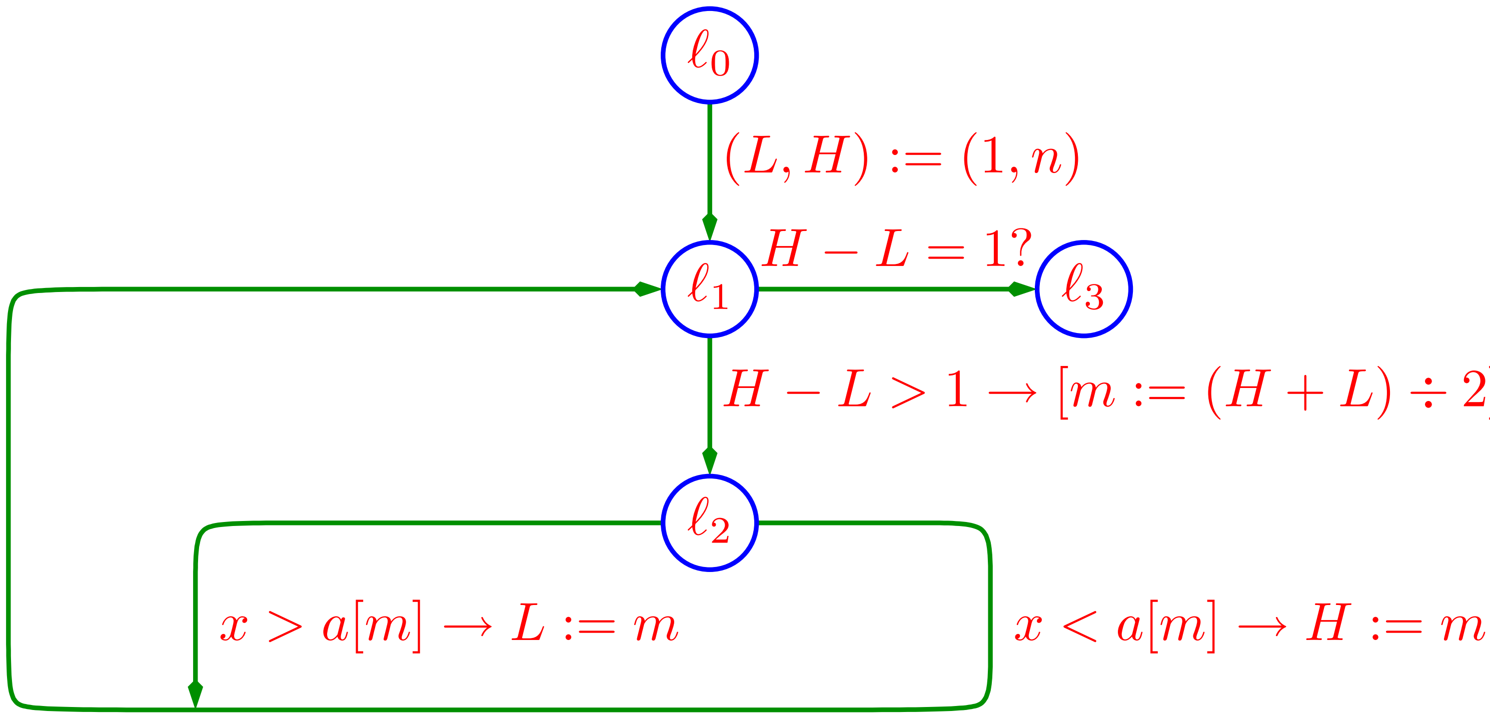
For a guarded command $\gamma : c \rightarrow [V := f(V)]$, we define the safety condition as $\Gamma_\gamma = \Gamma_c \ \wedge \ (c \rightarrow \Gamma_{V:=f(V)})$.

Let ℓ be a location in a program, and let $\gamma_1, \ldots, \gamma_k$ be the guarded commands labeling the edges departing from ℓ. Then, we define the safety condition for ℓ to be the conjunction $\Gamma_\ell : \quad \Gamma_{\gamma_1} \ \wedge \ \cdots \ \wedge \ \Gamma_{\gamma_k}$.

To prove that a program is p-fault free, we can use the proof method of Claim 9 where we replace the exit condition E_ℓ by the safety condition Γ_ℓ.

Example: Binary Search

Reconstider the binary search program.



The only non-trivial safety condition is $\Gamma_{\ell_2} : 1 \leq m \leq n$. To prove p-fault-freedom for this program, we take the same assertion network as before. The verification condition for location ℓ_2 is given by

$$\underbrace{(1 \leq L < H \leq n) \ \wedge \ \cdots}_{\varphi_1} \wedge \underbrace{H - L > 1}_{c_\pi} \quad \rightarrow \quad \underbrace{1 \leq (H + L) \div 2 \leq n}_{\Gamma_{\ell_2}(f_\pi(V))}$$

which is obviously valid.

Finding Inductive Assertions

The most difficult task in the application of the inductive assertion method is the design of a set of inductive assertions. Due to theoretical considerations of undecidability and incompleteness, we know that there can be no algorithm for finding inductive assertions.

At best, we can present a set of useful heuristics which will work in some of the cases.

It is useful to partition the assertion construction heuristics into two classes: bottom-up techniques and top-down heuristics.

Bottom-up techniques analyze the program without considerations of its specification. We usually infer first some invariant for an innermost loop and then use propagation techniques to export this invariant to other parts of the program.

In top down technique, we assume some invariant to be known outside of a loop (such as the post-condition part q of the specification) and proceed to infer an invariant for a point inside the loop.

Propagation Techniques

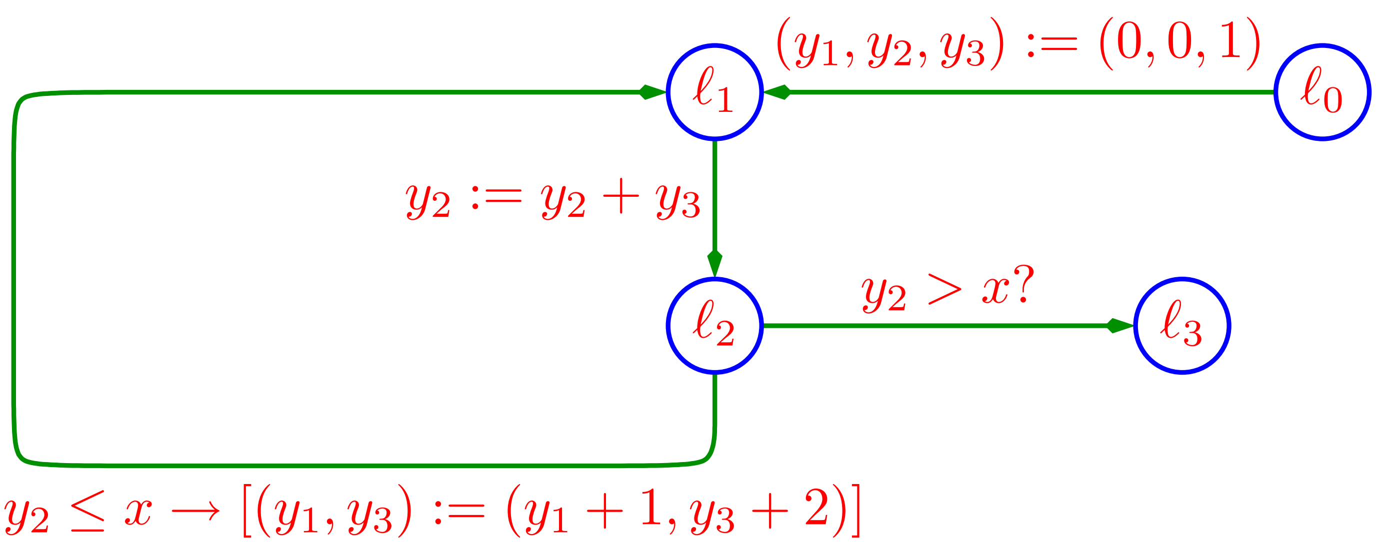
Let S be a set of locations which have already been assigned assertions, $\{\varphi_\ell \mid \ell \in S\}$, and $\widetilde{\ell} \notin S$ be an unassigned location. We can propagate assertions from S to $\widetilde{\ell}$ either backwards or forwards.

Backwards propagation

Let $\Pi_{\widetilde{\ell},S}$ be a set of paths connecting $\widetilde{\ell}$ to locations in S. The assertion propagated backwards from $\mathcal{S}$ to $\widetilde{\ell}$ is given by

$$pre(\widetilde{\ell}, S): \quad \bigwedge_{\pi \in \Pi_{\widetilde{\ell},S}} \Big(c_\pi(V) \rightarrow \varphi_{dest(\pi)}(f_\pi(V))\Big)$$

Consider for example program INT-SQUARE



in which $\varphi_2 = y_1^2 \leq x \ \wedge \ y_2 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1$. We can propagate φ_2 backwards towards ℓ_1 and obtain:

$$\varphi_1: \quad y_1^2 \leq x \ \wedge \ y_2 + y_3 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1$$

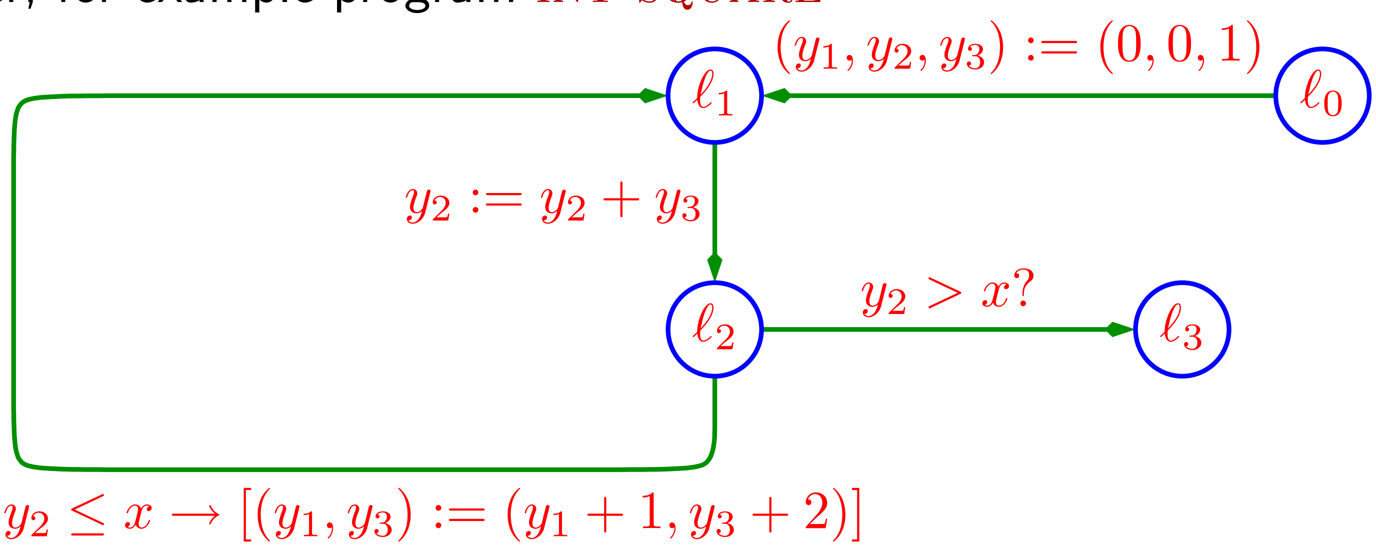
Forward Propagation

Let $\Pi_{S,\widetilde{\ell}}$ be a set of paths connecting locations in S to $\widetilde{\ell}$. The assertion propagated forwards from $\mathcal{S}$ to $\widetilde{\ell}$ is given by

$$post(S, \widetilde{\ell}) : \quad \bigvee_{\pi \in \Pi_{S,\widetilde{\ell}}} \exists \overline{V} : \left(\varphi_{srce(\pi)}(\overline{V}) \ \land \ c_\pi(\overline{V}) \ \land \ V = f_\pi(\overline{V})\right)$$

For the case that the function f_π is invertible, we can replace the disjunct $\exists \overline{V} : \left(\varphi_{srce(\pi)}(\overline{V}) \ \land \ c_\pi(\overline{V}) \ \land \ V = f_\pi(\overline{V})\right)$ by $\varphi_{srce(\pi)}(f_\pi^{-1}(V)) \land c_\pi(f_\pi^{-1}(V))$.

Consider, for example program INT-SQUARE



where $\varphi_0 : x \geq 0$ and $\varphi_2 : y_1^2 \leq x \ \land \ y_2 = (y_1+1)^2 \ \land \ y_3 = 2y_1 + 1$. Propagating forwards from $S = \{\ell_0, \ell_2\}$ to ℓ_1, we obtain

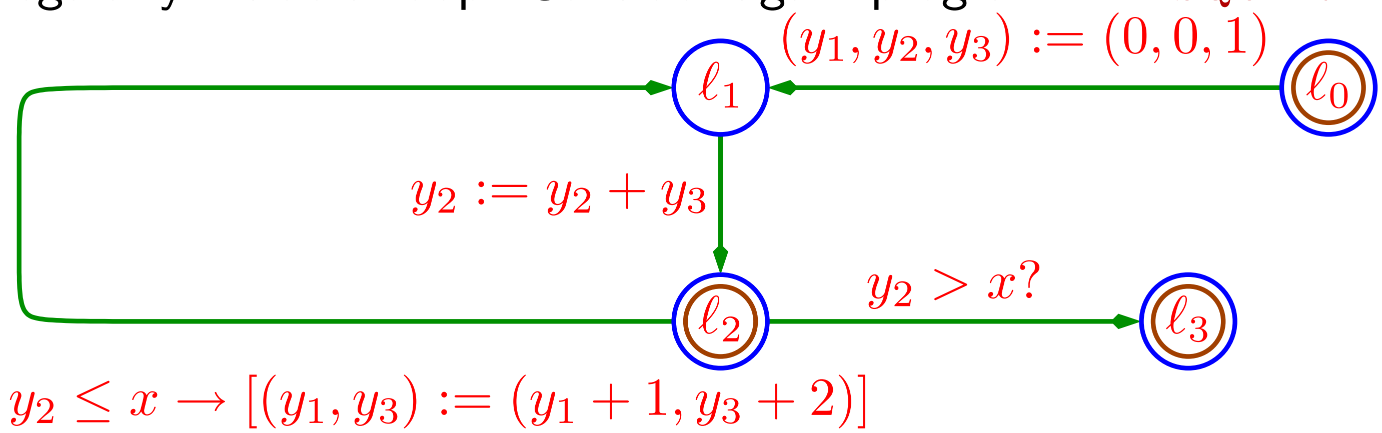
$$\varphi_1 : \quad \left(\begin{array}{ll} & x \geq 0 \ \land \ (y_1, y_2, y_3) = (0, 0, 1) \\ \lor & y_2 \leq x \ \land \ (y_1 - 1)^2 \leq x \ \land \ y_2 = y_1^2 \ \land \ y_3 - 2 = 2(y_1 - 1) + 1 \end{array}\right)$$

which can be simplified to

$$\varphi_1 : \quad y_1^2 \leq x \ \land \ y_2 = y_1^2 \ \land \ y_3 = 2y_1 + 1$$

Recurrence Equations

A useful bottom-up technique forms recurrence equations for variables which are modified regularly inside a loop. Consider again program INT-SQUARE.



Let $y_1(n)$ denote the value of variable y_1 at location ℓ_2 after the n'th iteration of the $\ell_2 \rightarrow \ell_1 \rightarrow \ell_2$ loop. Observing the way y_1 is modified during execution of this loop, we obtain the equation:

$$y_1(n+1) = y_1(n) + 1$$

which can be solved to obtain $y_1(n) = y_1(0) + n$. Since $y_1(0)$, the value of y_1 on the first visit to ℓ_2, is 0, we can conclude

$$y_1(n) = n \qquad (4)$$

The recurrence equation for $y_3(n)$ is $y_3(n+1) = y_3(n) + 2$. Since $y_3(0) = 1$, we can conclude

$$y_3(n) = 2n + 1 \qquad (5)$$

Finally, the recurrence equation for y_2 is $y_2(n+1) = y_2(n) + y_3(n+1)$. Since $y_2(0) = 1$, we can use Formula (3) to obtain

$$y_2(n) = 1 + \sum_{i=1}^{n} y_3(i) = 1 + \sum_{i=1}^{n} (2i+1) = (n+1)^2$$

Eliminating n between the expressions for $y_1(n)$, $y_2(n)$, and $y_3(n)$, we obtain

$$y_3 = 2y_1 + 1 \ \wedge \ y_2 = (y_1 + 1)^2.$$

Simultaneous Incrementation

In some cases, we can identify two variables y_1 and y_2 such that the overall effect of their modification within a loop can be summarized by the multiple assignment

$$(y_1, y_2) := (y_1 + c_1, y_2 + c_2)$$

It is obvious that we can form recurrence equations for such variables whose solution will be given by

$$y_1(n) = y_1(0) + nc_1 \qquad \text{and} \qquad y_2(n) = y_2(0) + nc_2$$

From these, we can infer the invariant:

$$\frac{y_1 - y_1(0)}{c_1} = \frac{y_2 - y_2(0)}{c_2}$$

For example, in the case of program INT-SQUARE, we can use this approach to obtain the invariant

$$\frac{y_1 - 0}{1} = \frac{y_3 - 1}{2}$$

which leads to the assertion $y_3 = 2y_1 + 1$ at location ℓ_2.

Simultaneous Multiplication

In other cases, we can identify two variables y_1 and y_2 such that the overall effect of their modification within a loop can be summarized by the multiple assignment

$$(y_1, y_2) := (c \cdot y_1, c \cdot y_2)$$

It is obvious that we can form recurrence equations for such variables whose solution will be given by

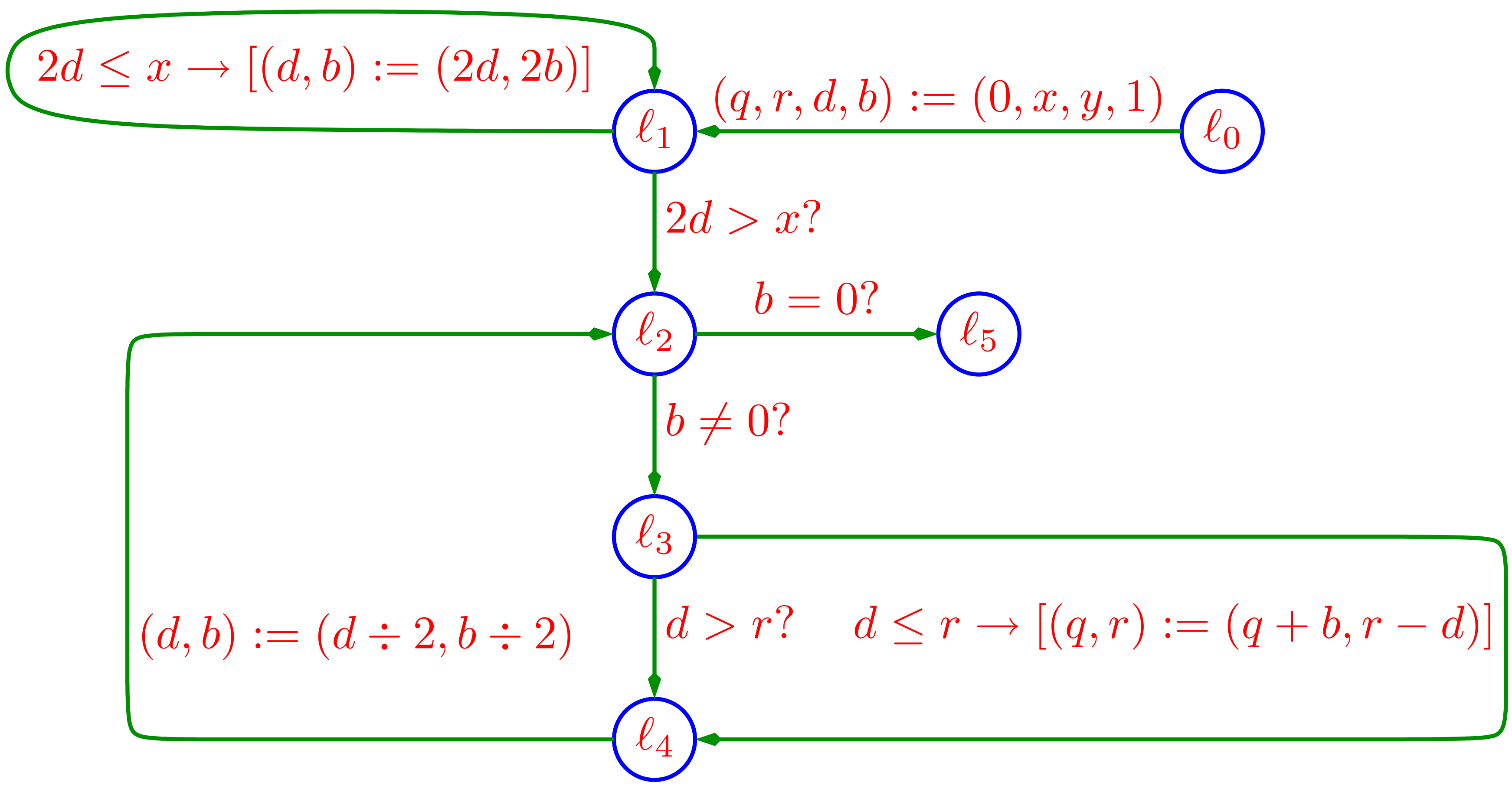
$$y_1(n) = y_1(0) \cdot c^n \qquad \text{and} \qquad y_2(n) = y_2(0) \cdot c^n$$

From these, we can infer the invariant:

$$\frac{y_1}{y_1(0)} = \frac{y_2}{y_2(0)}$$

Example: Integer Division

The following program divides the natural number $x \geq 0$ by the natural $y > 0$



The specification of this program is given by $\langle \varphi, \psi \rangle$, where

$$\begin{aligned} \varphi : &\quad x \geq 0 \ \wedge \ y > 0 \\ \psi : &\quad x = qy + r \ \wedge \ 0 \leq r < y \end{aligned}$$

We can apply the simultaneous multiplication heuristic to variables b and d in the loop of ℓ_1. As $b(0) = 1$ and $d(0) = y$, we obtain the invariant:

$$d = b \cdot y$$

A similar invariant holds at locations ℓ_3 and ℓ_4, but its proof is more involved due to the integer division.

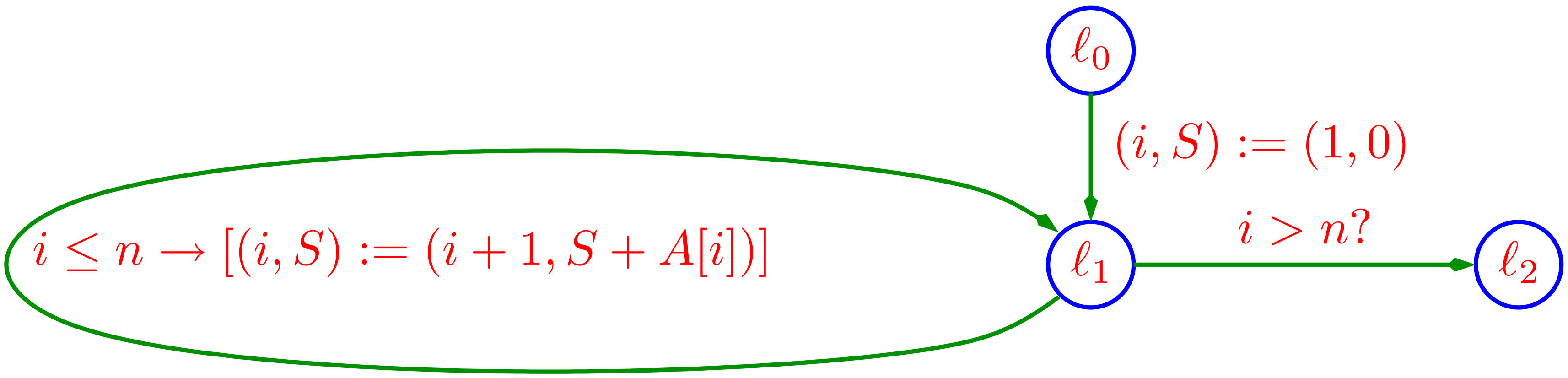
Top Down Techniques: Splitting Conjunctions

A useful top down technique is that of splitting conjunctions. It is often the case that, on exit from a loop, we expect to satisfy a conjunction $p \ \wedge \ q$. A valuable heuristic identifies one of the conjuncts (say p) as an assertion which will be maintained as a loop invariant, while the other conjunct q will be established by the exit condition.

For example, on exit from the INT-SQUARE program, the required post-condition is given by the conjunction $y_1^2 \leq x \ \wedge \ x < (y_1 + 1)^2$. It is feasible to split this conjunction into the assertion $y_1^2 \leq x$ which is maintained as an invariant of the loop in this program, and the assertion $x < (y_1 + 1)^2$ which is established when the exit condition becomes true.

Example: Array Summation

Consider the following program ARRAY-SUM, which sums the elements of an array $A[1..n]$.



The specification of this program is given by $\langle p, q\rangle$, where

$$p: \quad n \geq 0$$
$$q: \quad S = \sum_{j=1}^{n} A[j]$$

As it is given, the post-condition q is not a conjunction. However, it is possible to rewrite it, adding some more information, as

$$\widetilde{q}: \quad i = n+1 \ \wedge \ S = \sum_{j<i} A[j]$$

In this form, we can split $\widetilde{q}$ into the conjunct $S = \sum_{j<i} A[j]$ which is maintained throughout the loop of location ℓ_1, and the conjunct $i = n+1$ which is achieved on exit.