Sequential Programs Verification and Analysis

Amir Pnueli

Course: G22.3033-014

Wednesdays, 5-7 PM

Copies of presentations and Lecture Notes will be available at

http://www.cs.nyu.edu/courses/spring03/G22.3033-014/index.htm

Recommended textbooks:

- Program Verification by N. Francez, Addison-Wesley, 1992.
- Verification of Sequential and Concurrent Programs, by K.R. Apt and E.-R. Olderog, Spinger-Verlag, 1991.

Verification of Sequential Programs

In this course we will study methods for the formal verification of sequential programs. What is the interest in sequential verification?

- Historically, formal verification started with the study of sequential programs. Floyd's seminal paper [Flo67] defined the problem, and outlined the main principles of its solution: using invariants for proving partial correctness, and well-founded ranking functions to establish termination.
- These basic principles underly all subsequent developments in formal verification, including their extensions to reactive and parallel verification, methods of simulation and abstraction, and verification of functional programs.
- We use these principles in our work on translator validation.
- Recently, there has been a revival of interest in sequential verification, through encouragement of the use of assertions within programs, and intense activity in program analysis.

Partial List of Topics that Will be covered

- Programs and their specification.
- Various notions of correctness: partial correctness, termination, absence of failures.
- Using invariants for proving partial correctness.
- Using ranking functions for proving termination.
- Programs with procedures.
- Functional programs and their verification.
- Structured proof systems: Hoare logic, weakest precondition.
- Dealing with abstract data types and pointer structures.
- Abstract interpretation and program analysis.

The Verification Framework

The subject deals with relations of objects in two description languages on different levels:

- A programming language $\mathcal{P}$. Can be compiled and executed on conventional computing systems.
- A specification language $\mathcal{S}$. A higher level non-procedural language which offers a natural vehicle for humans to represent requirements and specification of computing tasks.

Questions which can be Asked

Given a verification framework, there are several questions one could ask about relationship between object in these two languages:

- The Synthesis Problem: Given a specification $S \in \mathcal{S}$, construct a program $P \in \mathcal{P}$ which satisfies the specification.
- The Analysis Problem: Given a program $P \in \mathcal{P}$, find its corresponding description $S \in \mathcal{S}$.
- The Verification Problem: Given a specification $S \in \mathcal{S}$ and a program $P \in \mathcal{P}$, check whether they are compatible, i.e. whether P satisfies S.
- The Debugging Problem: Given a specification $S \in \mathcal{S}$ and a program $P \in \mathcal{P}$ known not to satisfy S, find a program $P' \in \mathcal{P}$ "close" to P, i.e., transform P into P', such that P' satisfies S.
- The Optimization Problem: Given a specification $S \in \mathcal{S}$ and a program $P \in \mathcal{P}$ satisfying S. Among all programs P' "close" to P and satisfying S, find the "best" program (i.e. maximizing some performance metric).

A central notion which appears in all of these questions is that of a program $P \in \mathcal{P}$ satisfying a specification $S \in \mathcal{S}$. For that reason, we should study the verification problem first.

In general, all of these problems are difficult, undecidable, and at best, intractable. However, if $\mathcal{S}$ and $\mathcal{P}$ are close enough, they may admit algorithmic solutions. For example, compilation can be viewed as a special case of synthesis.

Program Represented by Transition Graphs

Our first programming language will be based on transition graphs. We assume a set of typed program variables V.

A transition graph is a labeled directed graph such that:

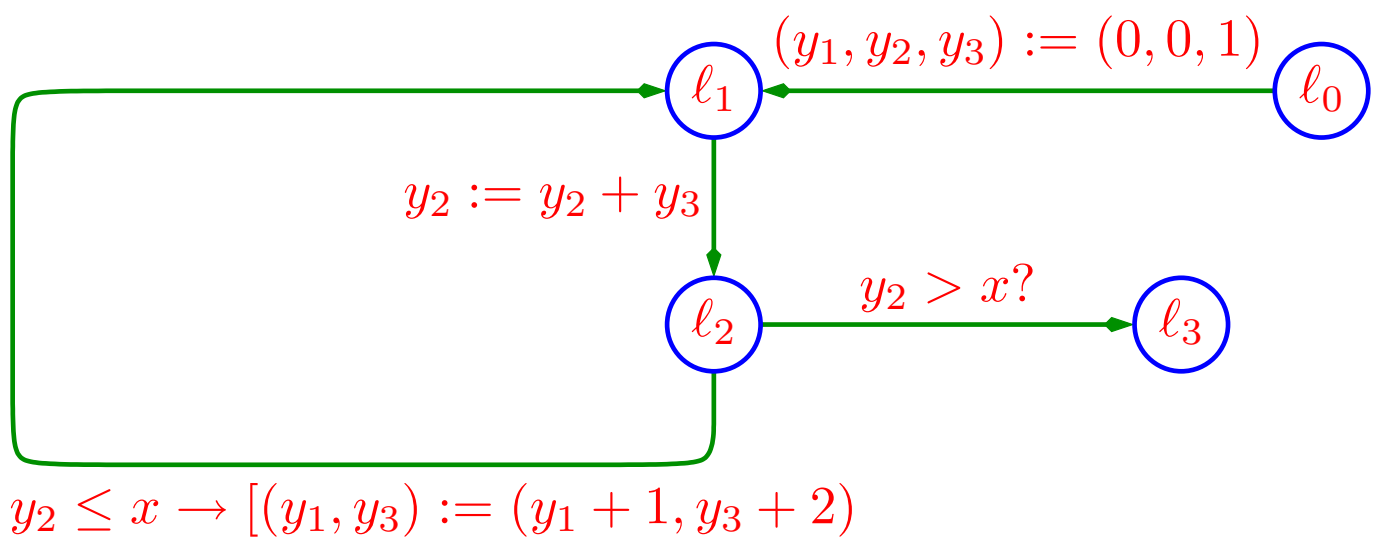
- All nodes are labeled by locations ℓ_i.
- There is one initial node, usually labeled by ℓ_0, and having no incoming edges.
- There is one terminal node, labeled ℓ_t with no outgoing edges.
- Nodes are connected by directed edges labeled by an instruction of the form

  $$c \rightarrow [\vec{y} := \vec{e}]$$

  where c is a boolean expression over V, $\vec{y} \subseteq V$ is a list of variables, and $\vec{e}$ is a list of expressions over V. In cases the assignment part is empty, we can abbreviate the label to a pure condition $c?$.
- Every node is on a path from ℓ_0 to ℓ_t.

Example: Integer Square Root Program

The following program INT-SQUARE computes in y_1 the integer square root of the input variable $x \geq 0$.



States and Computations

For simplicity, we assume that all program variables range over the same domain D. For example, for program INT-SQUARE, D is the domain of integers. We denote by $d = (d_1, \ldots, d_n)$ a sequence of D-values, which represent an interpretation (i.e., an assignment of values) of the program variables V.

A state of program P is a pair $\langle \ell, d \rangle$ consisting of a label ℓ and a data-interpretation d. A computation of program P is a maximal sequence

$$\sigma : \quad \langle \ell^0, d^0 \rangle, \langle \ell^1, d^1 \rangle, \ldots, \langle \ell_k, d^k \rangle \ \ldots,$$

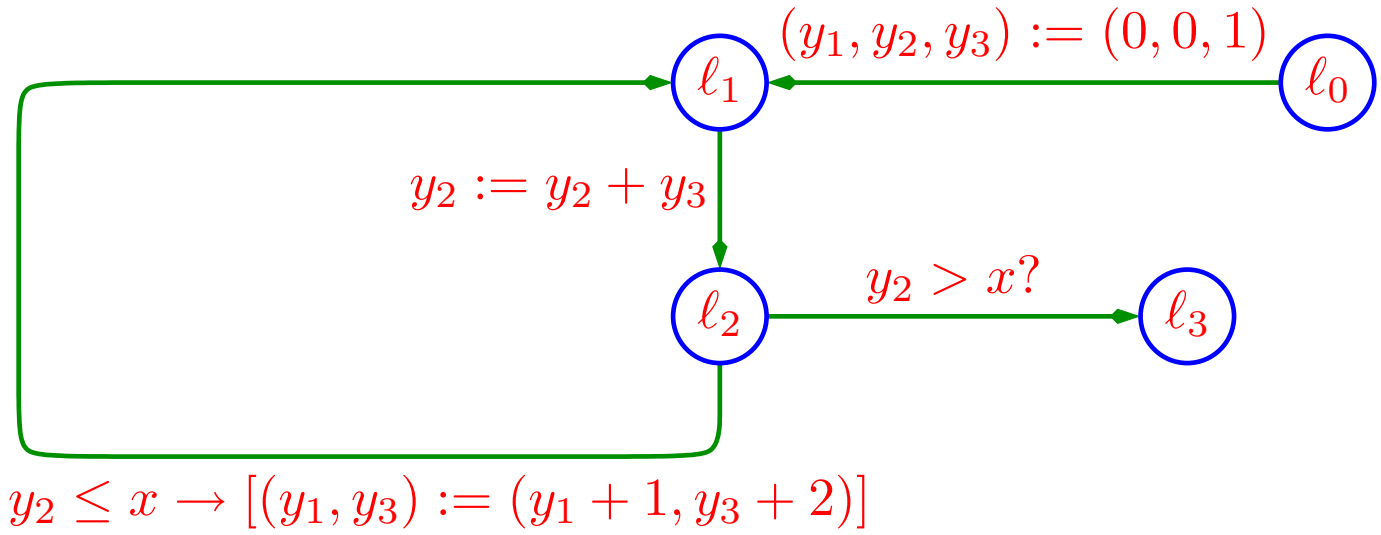
such that

- $\ell^0 = \ell_0$.
- For each $i = 0, 1, \ldots,$ there exists an edge connecting ℓ^i to ℓ^{i+1} and labeled by the instruction $c \rightarrow [\vec{y} := \vec{e}]$, such that $d^i \models c$ and $d^{i+1} = d^i$ **with** $\vec{y} := \vec{e}(d^i)$.

We denote by $\mathcal{C}omp(P, d)$ the set of computations of program P starting at data-state d.

An Example of a Computation

Reconsider program INT-SQUARE.



Following is a computation generated for $x = 5$:

$$\langle \ell_0; (-,-,-) \rangle,$$
$$\langle \ell_1; (0,0,1) \rangle, \quad \langle \ell_2; (0,1,1) \rangle, \quad \langle \ell_1; (1,1,3) \rangle, \quad \langle \ell_2; (1,4,3) \rangle,$$
$$\langle \ell_1; (2,4,5) \rangle, \quad \langle \ell_2; (2,9,5) \rangle, \quad \langle \ell_3; (2,9,5) \rangle$$

Results of Computations

Let σ be computation. We define the result of the computation σ, denoted $val(\sigma)$, according to the following cases:

- If the computation is finite, and the last state is $\langle \ell_t; d \rangle$, then $val(\sigma) = d$. We refer to such a computation as a terminating computation.
- If the computation is finite, and the last state is $\langle \ell; d \rangle$ for some $\ell \neq \ell_t$, we say that the computation fails and write $val(\sigma) = fail$. This is possible if all guards on edges departing from location ℓ are false on d. In particular if there are no edges departing from ℓ.
- If the computation is infinite, we say that the computation diverges, and write $val(\sigma) = \perp$.

For a program P and initial data-state d, we define the meaning of the program P as a function:

$$M(P, d) = \{val(\sigma) \mid \sigma \in \mathcal{C}omp(P, d)\}$$

It is customary to write $M(P, d)$ as $M[P](d)$ to emphasize that M is a mapping which, for each program P yields a function $M[P]$ of the type:

$$M[P] : D^n \mapsto 2^{D^n \cup \{fail, \perp\}}$$

Specifications

A specification for a sequential program is given by a pair (φ, ψ) of first-order formulas, where

- The pre-condition φ imposes constraints on the initial data state by which proper computations could start.
- The post-condition ψ specifies the properties the terminal data state of a proper computation should satisfy.

For example, a specification for program INT-SQUARE can be given by the pair

$$(x \geq 0, \qquad y_1^2 \leq x < (y_1 + 1)^2)$$

According to this specification, on initiation x should have a non-negative value while, on termination y_1 should be such that its square does not exceed x, but the square of $y_1 + 1$ should exceed x.

A computation whose initial state satisfies φ is called a φ-computation.

Correctness Statements

Given a specification (φ, ψ), we can formulate several notions of correctness.

- Partial Correctness. Program P is partially correct with respect to the specification (φ, ψ) if every terminating φ-computation ends in a ψ-state, i.e.

$$\varphi(d_0) \ \wedge \ d \in M[P](d_0) \quad \rightarrow \quad \psi(d)$$

- Success. A program is successful under φ (φ-successful) if there are no failing φ-computations. That is,

$$\varphi(d_0) \quad \rightarrow \quad \mathit{fail} \notin M[P](d_0)$$

- Convergence. A program is convergent under φ (φ-convergent, φ-terminating) if there are no divergent φ-computations. That is,

$$\varphi(d_0) \quad \rightarrow \quad \perp \notin M[P](d_0)$$

- Total Correctness. Program P is totally correct with respect to (φ, ψ) if it is partially correct, successful, and convergent under (φ, ψ).

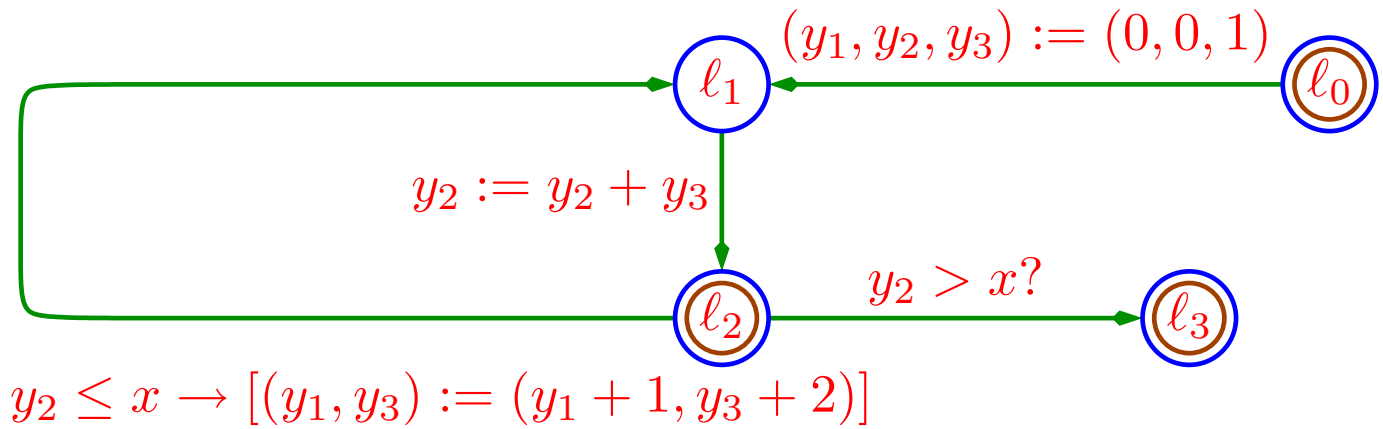
Proving Partial Correctness

We now present a proof method for proving partial correctness of a program. This proof method is called the method of inductive assertions [Flo67].

Step 1: Identifying a Cut-point Set

A cut-point set is a subset of locations $\mathcal{C} \subseteq \mathcal{L}$ such that $\ell_0, \ell_t \in \mathcal{C}$ and every cycle in the program's graph contains at least one cut-point (a member of $\mathcal{C}$).

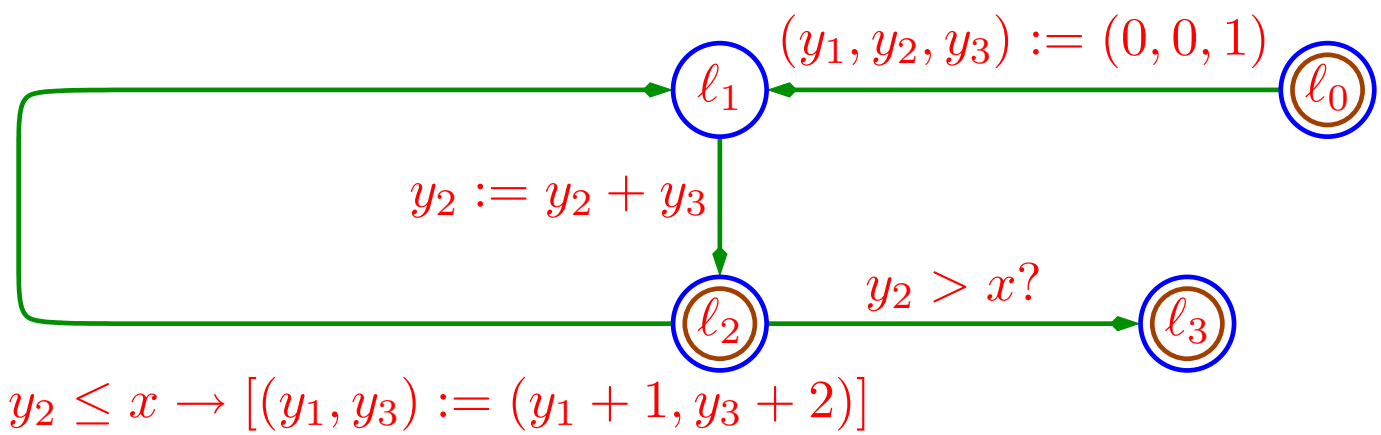
For example, for program INT-SQUARE, we can choose the cut-point set $\mathcal{C} = \{\ell_0, \ell_2, \ell_3\}$.



Step 2: Verification Paths

A verification path is a path from one cut-point to another cut-point, which does not pass through any other cut-point.

For example, in program INT-SQUARE, we have 3 verification paths.

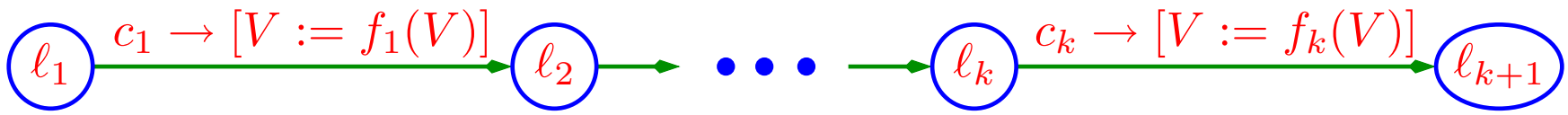


The verification paths for this program are given by

$$\begin{array}{lcl} \pi_{02} & : & \ell_0, \ell_1, \ell_2 \\ \pi_{22} & : & \ell_2, \ell_1, \ell_2 \\ \pi_{23} & : & \ell_2, \ell_3 \end{array}$$

Summary Guarded Commands

Consider a verification path π where, for simplicity, all assignments are made to the full set of program variables V.



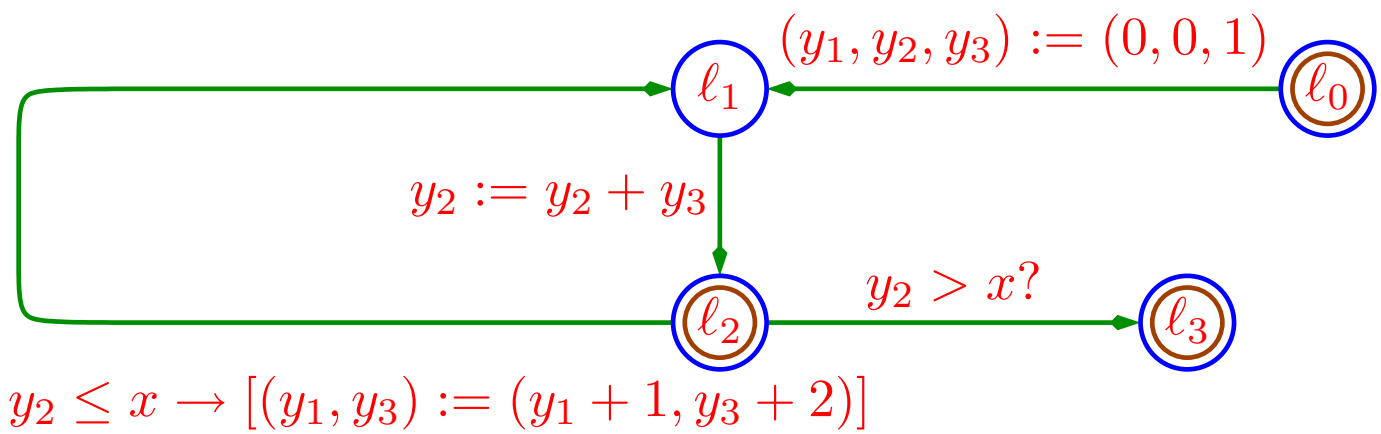
For such a path we can compute a traversal condition c_π and a data transformation f_π. Condition c_π when satisfied at ℓ_1 guarantees that it is possible to traverse the path π. The transformation f_π specifies the values of V at the end of an execution of π as a function of the values of V in the beginning of such execution. They are respective given by:

$$\begin{array}{lcl} c_\pi & : & c_1(V) \wedge c_2(f_1(V)) \wedge \cdots \wedge c_k(f_{k-1}(\cdots f_1(V) \cdots)) \\ f_\pi & : & f_k(f_{k-1}(\cdots f_2(f_1(V)) \cdots)) \end{array}$$

Given these constructs we can summarize the effect of executing the path π by the summary guarded command $G_\pi : c_\pi \rightarrow [V := f_\pi(V)]$.

Application to INT-SQUARE

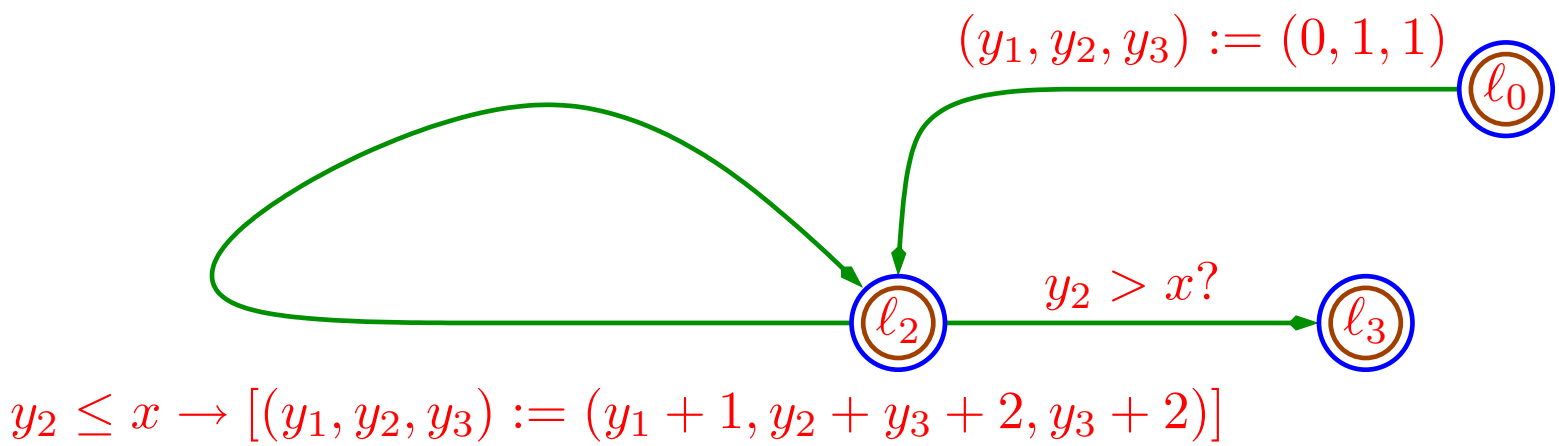
Apply this procedure to program INT-SQUARE.



The summary guarded commands for the 3 verification paths are given by:

$$
\begin{array}{lcl}
G_{02} & : & (y_1, y_2, y_3) := (0, 1, 1) \\
G_{22} & : & y_2 \leq x \rightarrow [(y_1, y_2, y_3) := (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)] \\
G_{23} & : & y_2 > x \rightarrow [(y_1, y_2, y_3) := (y_1, y_2, y_3)]
\end{array}
$$

Once we derive these summary guarded commands, it is possible to construct the following reduced version of the original program.

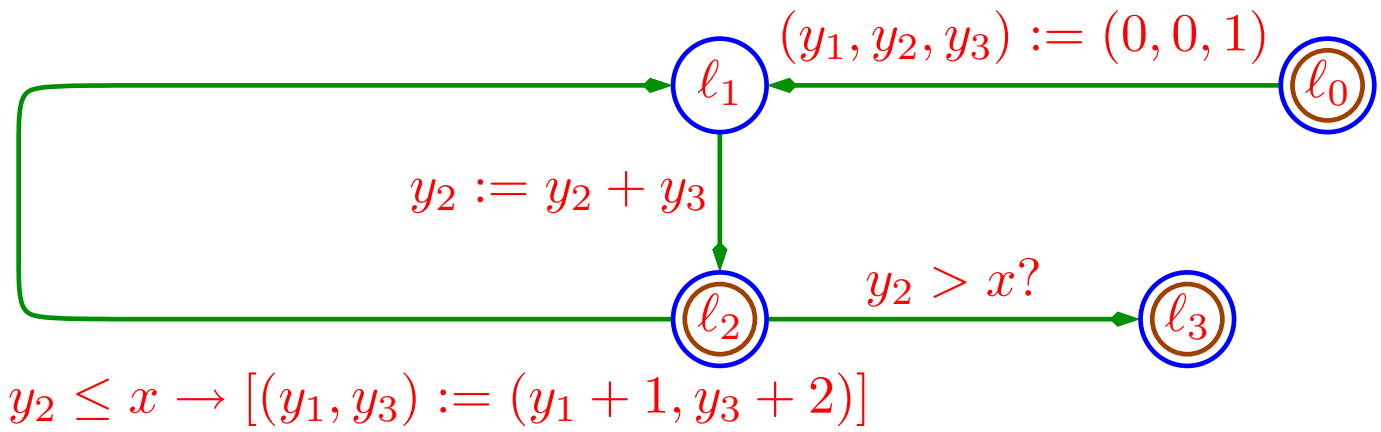


This reduced program is weakly equivalent to the original program in the sense that it preserves all successful terminating computations and all divergent computations. However, it may lose some failing computations of the original program.

Step 3: Devise an Assertion Network

With each cut-point $\ell_i \in \mathcal{C}$ associate an assertion φ_i (first-order formula) over V.

For example, for program INT-SQUARE,



we can form the following assertion network:

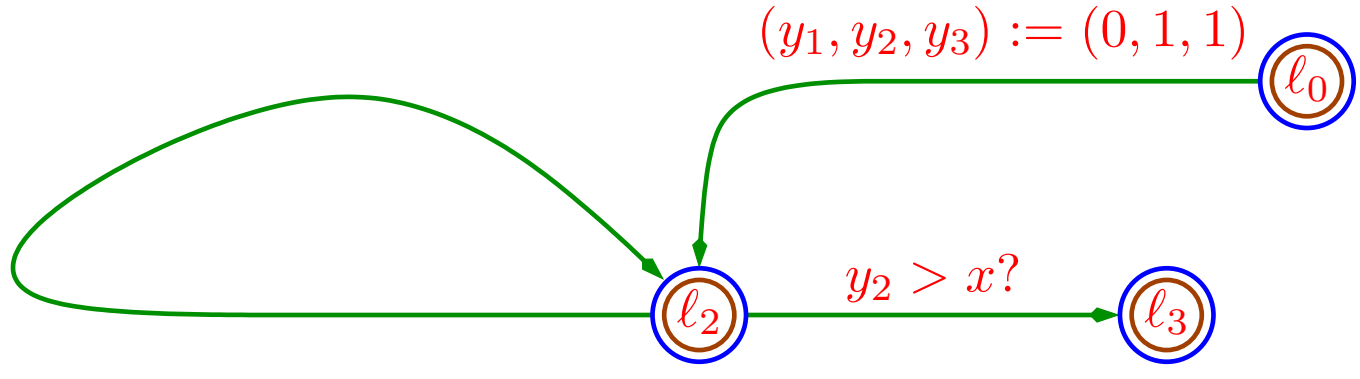
$$\begin{array}{lcl} \varphi_0 & : & x \geq 0 \\ \varphi_2 & : & y_1^2 \leq x \ \wedge \ y_2 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1 \\ \varphi_3 & : & y_1^2 \leq x < (y_1+1)^2 \end{array}$$

Step 4: Form Verification Conditions

For each verification path π connecting cut-point ℓ_i to cut-point ℓ_j, we form the verification condition

$$VC_\pi : \quad \varphi_i(V) \wedge c_\pi \quad \rightarrow \quad \varphi_j(f_\pi(V))$$

For example, for program INT-SQUARE



and the assertion network

$$\begin{array}{lcl} \varphi_0 & : & x \geq 0 \\ \varphi_2 & : & y_1^2 \leq x \ \wedge \ y_2 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1 \\ \varphi_3 & : & y_1^2 \leq x < (y_1+1)^2 \end{array}$$

we obtain the following set of verification conditions:

$$\begin{array}{lcl} VC_{02} & : & x \geq 0 \ \rightarrow \ 0^2 \leq x \ \wedge \ 1 = (0+1)^2 \ \wedge \ 1 = 2 \cdot 0 + 1 \\ VC_{22} & : & y_1^2 \leq x \ \wedge \ y_2 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1 \ \wedge \ y_2 \leq x \ \rightarrow \\ & & (y_1+1)^2 \leq x \ \wedge \ y_2 + y_3 + 2 = ((y_1+1)+1)^2 \ \wedge \ y_3 + 2 = 2(y_1+1) + 1 \\ VC_{23} & : & y_1^2 \leq x \ \wedge \ y_2 = (y_1+1)^2 \ \wedge \ y_3 = 2y_1 + 1 \ \wedge \ y_2 > x \ \rightarrow \\ & & y_1^2 \leq x < (y_1+1)^2 \end{array}$$

Inductive and Invariant Networks

An assertion network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ for a program P is said to be inductive if all the verification conditions VC_π for all verification paths π in P are valid.

Network $\mathcal{N}$ is said to be invariant if for every execution state $\langle \ell_i, d \rangle$ occurring in a φ_0-computation, where $\ell_i \in \mathcal{C}$, $d \models \varphi_i$. That is, on every visit of a φ_0-computation at a cut-point ℓ_i the visiting data state satisfies the corresponding assertion φ_i associated with ℓ_i.

Claim 1. *Every inductive network is invariant.*

Proof Let $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ be an inductive network. Let

$$\sigma: \quad \langle \ell_{i_0}, d_0 \rangle \xrightarrow{\pi_0} \langle \ell_{i_1}, d_1 \rangle \xrightarrow{\pi_1} \cdots \xrightarrow{\pi_{k-1}} \langle \ell_{i_k}, d_k \rangle \xrightarrow{\pi_k} \cdots$$

be a φ_0-computation where we explicitly display the sequence of cut-points $\ell_0 = \ell_{i_0}, \ell_{i_1}, \ldots$ visited by σ and the verification paths $\pi_0, \pi_1, \ldots$ connecting them.

We will prove by induction on $j = 0, 1, \ldots$ that $d_j \models \varphi_{i_j}$. For $j = 0$, we consider the cut-point $\ell_{i_0} = \ell_0$. Since σ is a φ_0-computation, we have that $d_0 \models \varphi_0$.

Assume now that $d_j \models \varphi_{i_j}$. We will show that $d_{j+1} \models \varphi_{i_{j+1}}$. Since σ proceeded from ℓ_{i_j} to $\ell_{i_{j+1}}$ through verification path π_j, we know that $d_j \models c_{\pi_j}$ and $d_{j+1} = f_{\pi_j}(d_j)$. We also have that the verification condition

$$VC_{\pi_j}: \quad \varphi_{i_j}(d_j) \ \wedge \ c_{\pi_j}(d_j) \quad \rightarrow \quad \varphi_{i_{j+1}}(f_{\pi_j}(d_j))$$

holds. Since both $\varphi_{i_j}(d_j)$ and $c_{\pi_j}(d_j)$ are true, we conclude that $\varphi_{i_{j+1}}(f_{\pi_j}(d_j)) = \varphi_{i_{j+1}}(d_{j+1})$ is also true. It follows that $d_{j+1} \models \varphi_{i_{j+1}}$. ∎

Consequences

From Claim 1 we conclude:

Corollary 2. *If $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ is an inductive network, then program P is partially correct with respect to the specification (φ_0, φ_t).*

Let (p, q) be a specification. We say that the network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ entails the specification (p, q) if the following two implications are valid:

$$p \rightarrow \varphi_0 \qquad\qquad \varphi_t \rightarrow q$$

Corollary 3. *If $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ is an inductive network which entails the specification (p, q), then program P is partially correct with respect to (p, q).*

This leads to the final formulation of the inductive assertion proof method.

In order to prove that program P is partially correct w.r.t specification (p, q), find an assertion network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_t\}$ and prove that $\mathcal{N}$ is inductive and that it entails the specification (p, q).