Temporal Annotations and Their Validation

Amir Pnueli

Weizmann Institute of Sciences and New York University

Grand Challenges of Informatics, Budapest, September, 2006

Based on Joint work with:

Zohar Manna Stanford Krishna Palem Georgia Tech

The VSTTE Vision

Reach the stage at which a program would be allowed to run only if it is syntactically correct, type safe, and semantically correct – the verifying compiler.

This dream is about 40 years old. Why should we believe that there is now a better chance of its realization?

- Impressive progress in the technologies of automated theorem proving, abstraction, and model checking.
- Great success in hardware verification, and its recent porting to software model checking.
- Complementary developments in program analysis which grew out of abstract interpretation and compiler technologies.
- Greater maturity of software engineers, who are increasingly ready to adopt formal practices if they are shown to offer worthy return against invested effort.

The Main Interaction Mode: Program Annotation

Many of the promising techniques are automatic. But some user interaction will always be required.

A major mode of user interaction is based on program annotation where the user annotates his program at various control points by formal assertions. The intended meaning of such annotation is that the assertion should hold whenever execution reaches the relevant control point.

Annotations can be used for different purposes in different modes:

- Deep Testing. While testing the program under different inputs, check that the assertions hold whenever execution visits their control points.
- Run-Time Verification. Compile the assertions into checks that are exercised during execution of the program, raising an alarm if violated. Strong optimization may remove some of these run-time checks.
- Static Verification. Formally verify that the assertions hold whenever visited by any execution.

Therefore

There may be some interest in

- Developing a more extensive theory of program annotation, including recursive procedures and termination.
- Considering the extension of annotations from simple state predicates to more general temporal assertions.

Floyd's Theory of Annotated Programs and some Extensions

Programs will be presented as transition graphs. We assume a set of typed program variables V.

A transition graph is a labeled directed graph such that:

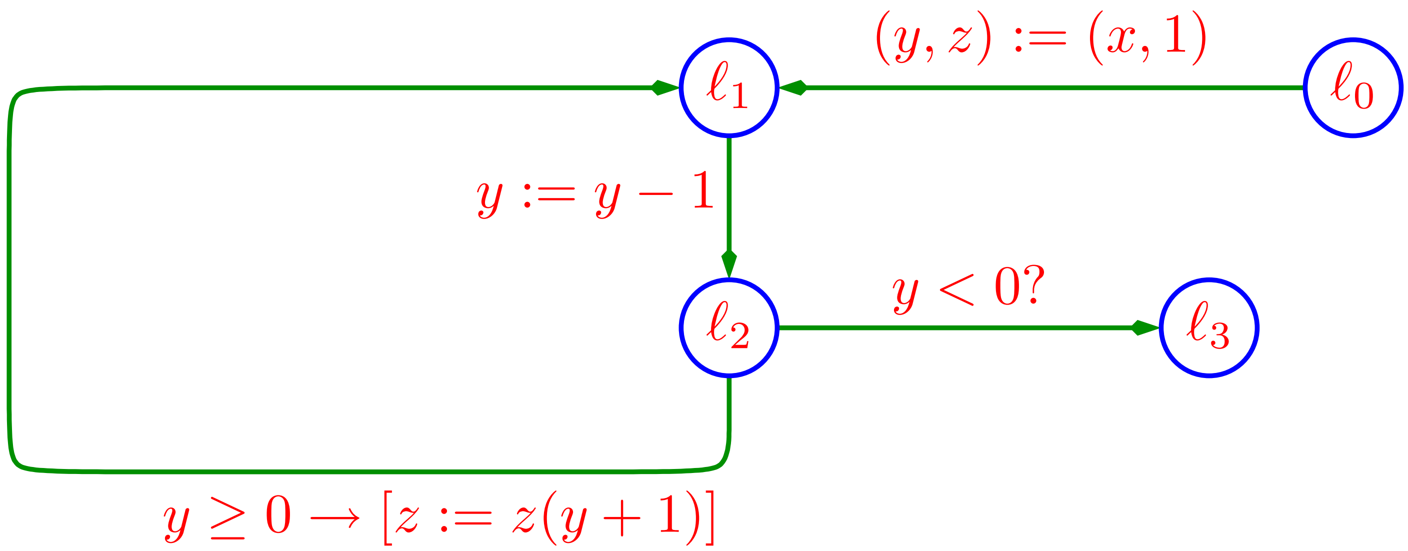
- All nodes are labeled by locations ℓ_i.
- There is one initial node, usually labeled by ℓ_0, and having no incoming edges.
- There is one terminal node, labeled ℓ_t with no outgoing edges.
- Nodes are connected by directed edges labeled by an instruction of the form

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

  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: Factorial Program

The following program FACTORIAL computes in z the factorial function $x!$ of the input variable $x \geq 0$.



Specifications

A specification for a sequential program is given by a pair (φ, ψ) of first-order formulas (assertions), 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 FACTORIAL can be given by the pair

$$(x \geq 0, \qquad z = x!)$$

According to this specification, on initiation x should have a non-negative value while, on termination z should equal $x!$.

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.
- Termination. A program is terminating under φ (φ-terminating) if there are no divergent φ-computations.
- Total Correctness. Program P is totally correct with respect to (φ, ψ) if it is partially correct w.r.t. (φ, ψ) and φ-terminating.

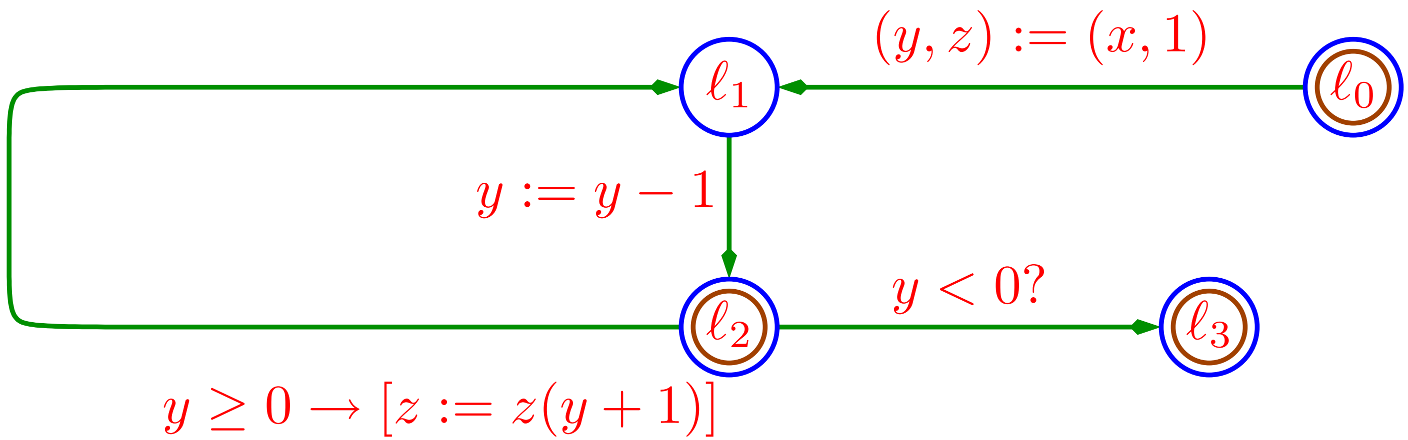
Proving Partial Correctness

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 $\mathcal{C}$).

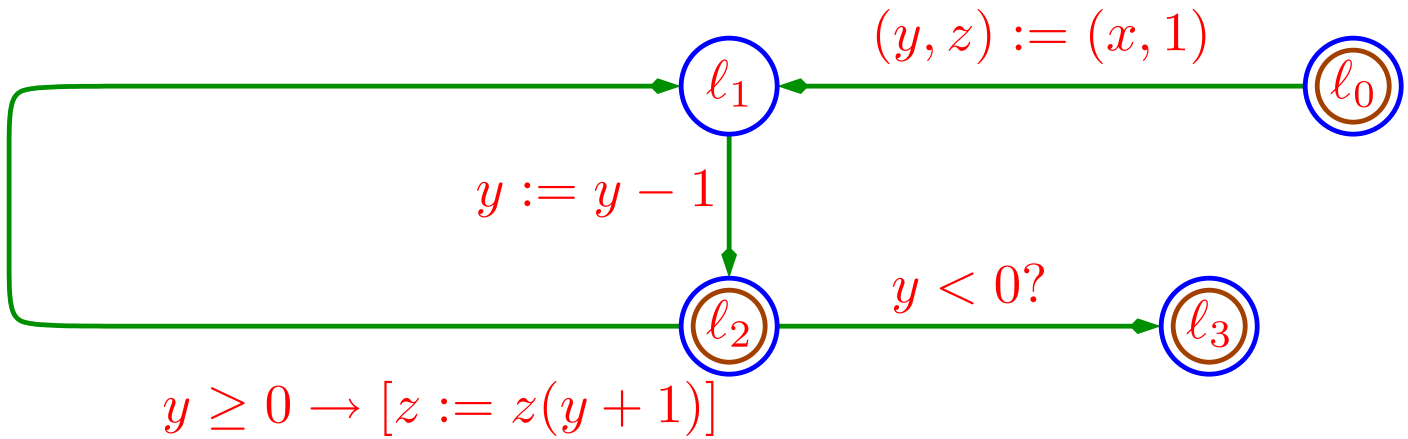
For example, for program FACTORIAL, we can choose the cut-point set $\mathcal{C} = \{\ell_0, \ell_2, \ell_3\}$.



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 FACTORIAL, we have 3 verification paths.

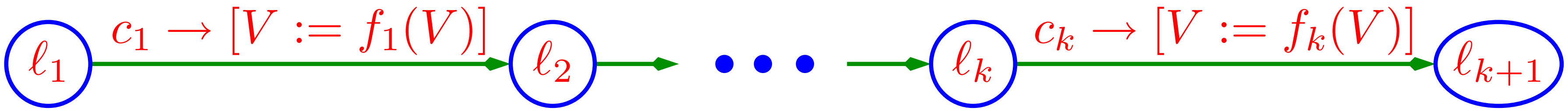


The verification paths for this program are given by

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

Summary Guarded Commands

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



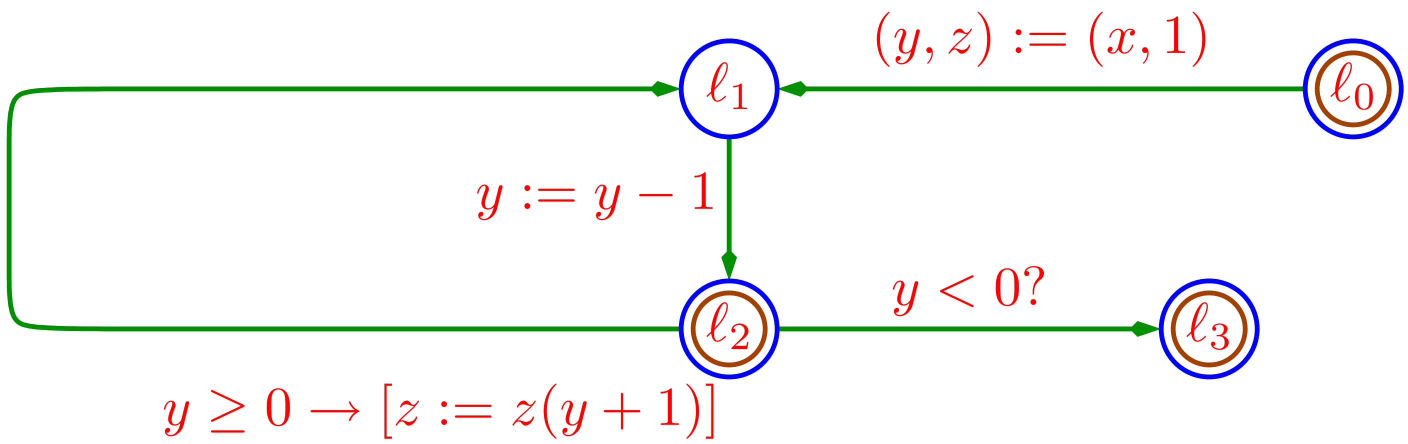
For such a path we can compute a traversal condition c_π and a data transformation f_π. Condition c_π when satisfied at ℓ_1 guarantees that it is possible to traverse the path π. The transformation f_π specifies the values of V at the end of an execution of π as a function of the values of V in the beginning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

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

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 FACTORIAL

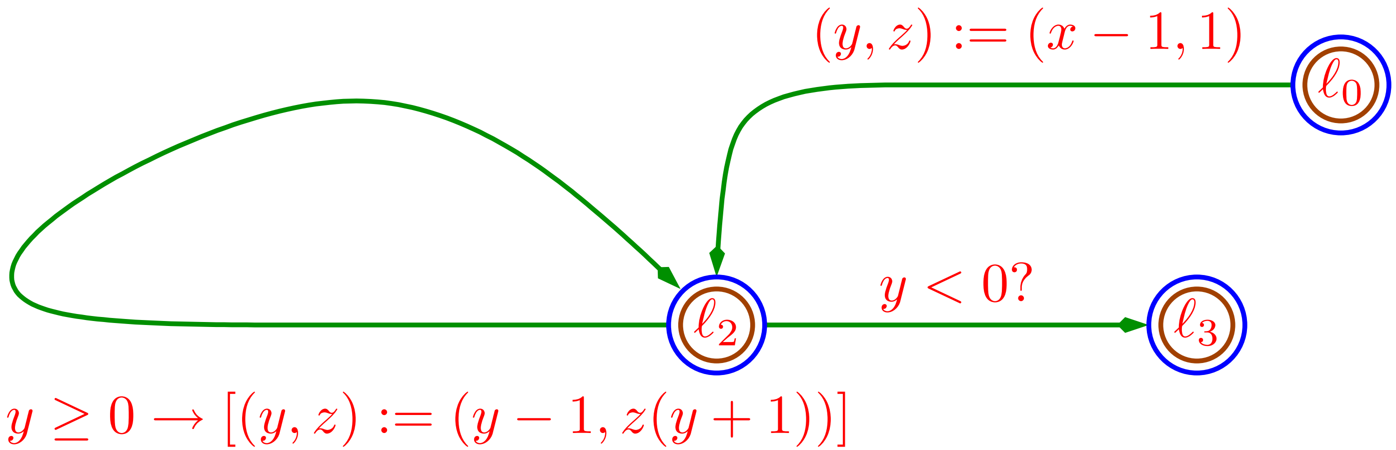
Apply this procedure to program FACTORIAL.



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

$$\begin{array}{lcl} G_{02} & : & (y, z) := (x - 1, 1) \\ G_{22} & : & y \geq 0 \rightarrow [(y, z) := (y - 1, z(y + 1))] \\ G_{23} & : & y < 0 \rightarrow [z := z] \end{array}$$

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

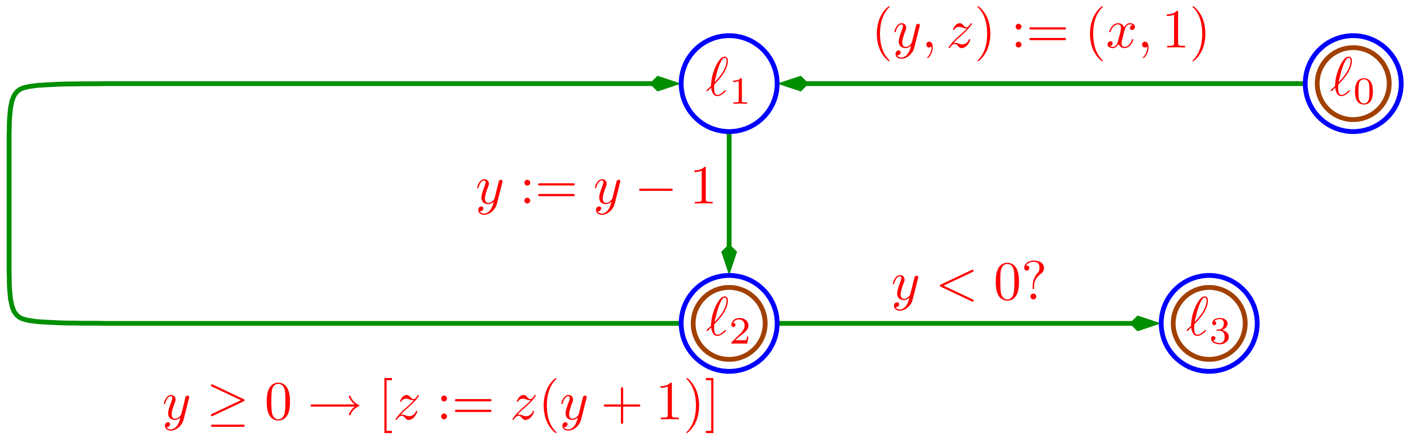


This reduced program is weakly equivalent to the original program in the sense that it preserves all successful terminating computations and all divergent computations. However, it may lose some failing computations of the original program.

Step 3: Devise an Assertion Network (Annotate)

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

For example, for program FACTORIAL,



we can form the following assertion network:

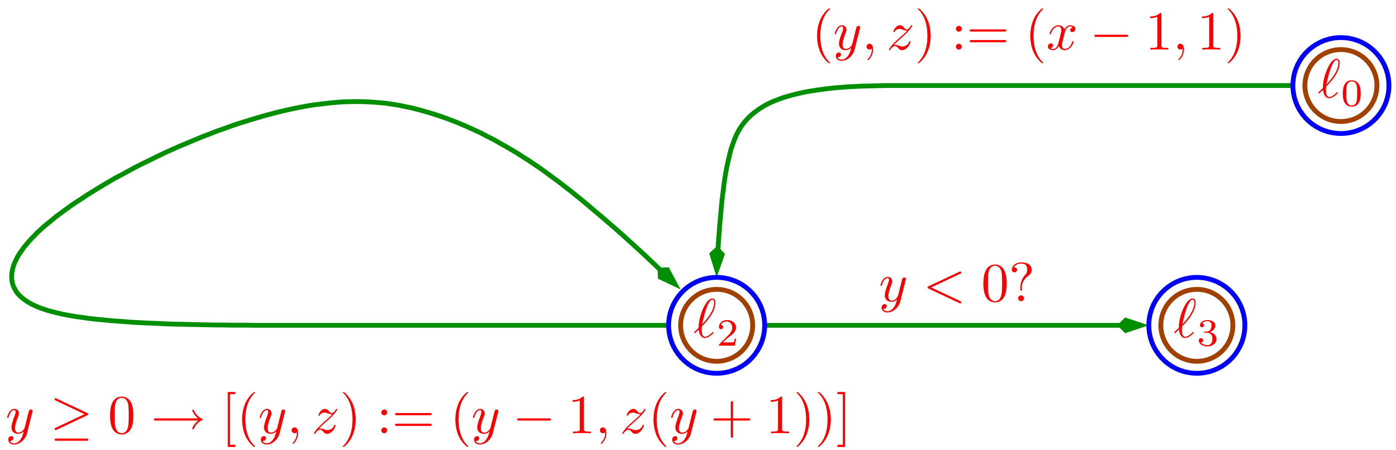
$$\begin{aligned} \varphi_0 &: \quad x \geq 0 \\ \varphi_2 &: \quad -1 \leq y < x \ \wedge \ x! = z \cdot (y+1)! \\ \varphi_3 &: \quad z = x! \end{aligned}$$

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 summarized program FACTORIAL



and the assertion network

$$\begin{array}{lcl} \varphi_0 & : & x \geq 0 \\ \varphi_2 & : & -1 \leq y < x \ \wedge \ x! = z \cdot (y+1)! \\ \varphi_3 & : & z = x! \end{array}$$

we obtain the following set of verification conditions:

$$
\begin{array}{lcl}
VC_{02} & : & x \geq 0 \quad \rightarrow \quad (-1 \leq x-1 < x) \ \wedge \ x! = 1 \cdot ((x-1)+1)! \\
VC_{22} & : & (-1 \leq y < x) \ \wedge \ x! = z \cdot (y+1)! \ \wedge \ y \geq 0 \quad \rightarrow \\
 & & \qquad (-1 \leq y-1 < x) \ \wedge \ x! = (z(y+1)) \cdot ((y-1)+1)! \\
VC_{23} & : & (-1 \leq y < x) \ \wedge \ x! = z \cdot (y+1)! \ \wedge \ y < 0 \quad \rightarrow \\
 & & \qquad z = x!
\end{array}
$$

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.*

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).

Dependence on the Cut-Set

The success of Floyd's method does not depend on the choice of the cut-set. A special case is that of a full cut-set $\mathcal{C} = \mathcal{L}$ in which the cut-set includes all the locations in the program.

The following claim shows that any inductive assertions which is not full, can be extended to a bigger inductive network.

Claim 4. [Inductive networks can be extended] *Let $\mathcal{N} = \langle \mathcal{C}, \{\varphi_\ell \mid \ell \in \mathcal{C}\}\rangle$ be an inductive assertion network, and $\widetilde{\ell} \notin \mathcal{C}$ a location not in $\mathcal{C}$. There exists an inductive assertion network over the extended cut-set $\widetilde{\mathcal{C}} = \mathcal{C} \cup \{\widetilde{\ell}\}$ which agrees with $\mathcal{N}$ on the assertions φ_ℓ for all $\ell \in \mathcal{C}$.*

Next, we show that it is also possible to remove cut-points, provided the remaining set is still a cut-set.

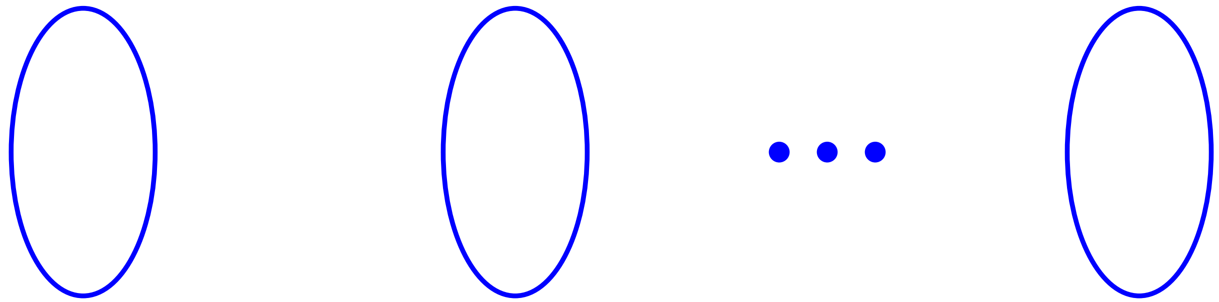
Claim 5. *Let $\mathcal{N} = \langle \mathcal{C}, \{\varphi_\ell \mid \ell \in \mathcal{C}\}\rangle$ be an inductive network. Let $\widetilde{\ell} \in \mathcal{C}$ be a location in $\mathcal{C}$ such that $\overline{\mathcal{C}} = \mathcal{C} - \{\widetilde{\ell}\}$ is a cut-set. Then the network $\overline{\mathcal{N}} = \langle \overline{\mathcal{C}}, \{\varphi_\ell \mid \ell \in \overline{\mathcal{C}}\}\rangle$, obtained by removing $\widetilde{\ell}$ and $\varphi_{\widetilde{\ell}}$ from $\mathcal{N}$, is also inductive.*

It follows that we can always move from network $\mathcal{N}_1$ to network $\mathcal{N}_2$, by completing $\mathcal{N}_1$ to a full network, and then removing all cut-points not in $\mathcal{N}_2$.

Extension to Procedures

We will now extend our treatment of programs to the consideration of programs with procedures. A program P in the extended language consists of $m+1$ modules: $P_0, P_1, \ldots, P_m$, where P_0 is the main module, and $P_1, \ldots, P_m$ are procedures which may be called from P_0 or from other procedures.

$P_0(\textbf{in } \vec{x}; \textbf{ out } \vec{z})$ $P_1(\textbf{in } \vec{x}; \textbf{ out } \vec{z})$ $P_m(\textbf{in } \vec{x}; \textbf{ out } \vec{z})$



Each module P_i is presented as a flow-graph with its own set of locations $\mathcal{L}_i = \{\ell^i_0, \ell^i_1, \ldots, \ell^i_t\}$. It must have ℓ^i_0 as its only entry point, ℓ^i_t as its only exit, and every other location must be on a path from ℓ^i_0 to ℓ^i_t.

The variables of each module P_i are partitioned into $\vec{y} = (\vec{x}; \vec{u}; \vec{z})$. We refer to $\vec{x}$,$\vec{y}$, and $\vec{z}$ as the input, working, and output variables, respectively. A module cannot modify its own input variables.

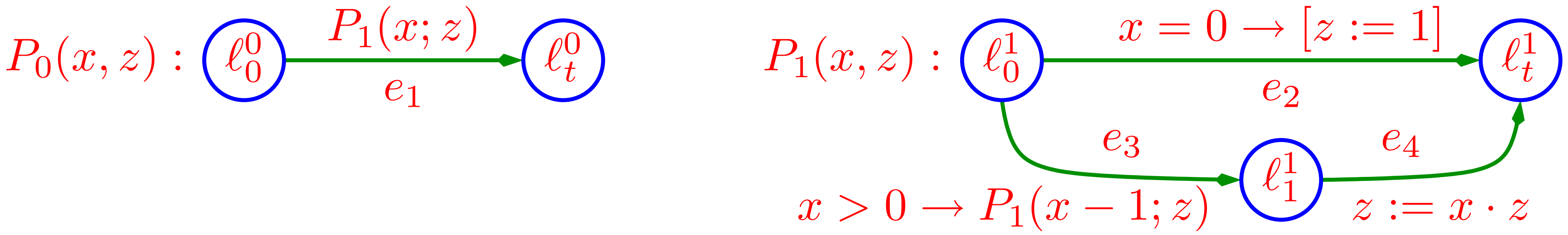
Instructions of Procedural Programs

Edges in the graph are labeled by an instruction which must be one of

- An assignment $c(\vec{y}) \to [\vec{v} := f(\vec{y})]$, where the left-hand side variables $\vec{v} \subseteq \{\vec{u}, \vec{z}\}$ may not include any member of $\vec{x}$.
- A procedure call $c(\vec{y}) \to P_j(\vec{e}; \vec{v})$, where $\vec{e}$ is a list of expressions over $\vec{y}$, and $\vec{v} \subseteq \{\vec{u}, \vec{z}\}$ is a list of distinct variables not including any member of $\vec{x}$. We refer to $\vec{e}$ and $\vec{y}$ as the actual arguments of the call.

Example: Factorial

Consider the following program for computing the factorial of a natural number.



Following is a computation of this program for input $x = 3$:

$$\langle \ell_0^0;\ (3, \bot) \rangle \xrightarrow{e_1}$$
$$\langle \ell_0^1;\ (3, \bot) \rangle \xrightarrow{e_3}$$
$$\langle \ell_0^1;\ (2, \bot) \rangle \xrightarrow{e_3}$$
$$\langle \ell_0^1;\ (1, \bot) \rangle \xrightarrow{e_3}$$
$$\langle \ell_0^1;\ (0, \bot) \rangle \xrightarrow{e_2} \langle \ell_t^1;\ (0, 1) \rangle \xrightarrow{\textit{return}}$$
$$\langle \ell_1^1;\ (1, 1) \rangle \xrightarrow{e_4} \langle \ell_t^1;\ (1, 1) \rangle \xrightarrow{\textit{return}}$$
$$\langle \ell_1^1;\ (2, 1) \rangle \xrightarrow{e_4} \langle \ell_t^1;\ (2, 2) \rangle \xrightarrow{\textit{return}}$$
$$\langle \ell_1^1;\ (3, 2) \rangle \xrightarrow{e_4} \langle \ell_t^1;\ (3, 6) \rangle \xrightarrow{\textit{return}}$$
$$\langle \ell_t^0;\ (3, 6) \rangle$$

Proving Partial Correctness

We extend the inductive assertion method to deal with procedural programs. A cut-set $\mathcal{C}$ is a set of locations in $\mathcal{L} = \mathcal{L}_0 \cup \cdots \cup \mathcal{L}_m$ such that:

1. Every loop in each P_i, $i = 0, \ldots, m$ contains at least one location of $\mathcal{C}$.
2. For every $i = 0, \ldots, m$, both ℓ_0^i and ℓ_t^i belong to $\mathcal{C}$.
3. For every edge $\ell_i \xrightarrow{e} \ell_j$ labeled by a procedure call, both ℓ_i and ℓ_j are in $\mathcal{C}$.

An assertion network associates an assertion $\varphi_i^j(\vec{y})$ with each location ℓ_i^j. For each module P_k, we denote φ_0^k by p_k and require that $p_k = p_k(\vec{x})$ depends only on the input variables of the module. Similarly, we denote φ_t^k by q_k and require that $q_k = q_k(\vec{x}; \vec{z})$ depends only on the input and output variables of the module.

The input predicate $p_k(\vec{x})$ imposes constraints on the input variables we expect on entry to module P_k. The output predicate $q_k(\vec{x}; \vec{z})$ specifies the relation between the output results and the input values.

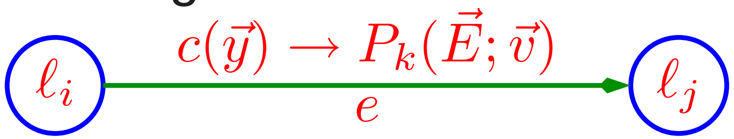
The Verification Conditions

We consider two types of verification conditions.

Let π be a verification path leading from location ℓ_i to location ℓ_j such that all edges in π are labeled by guarded assignment instructions. We refer to such a path as an assignment path. As usual, let c_π denote the traversal condition for π, and let $\vec{y} := f_\pi(\vec{y})$ summarize the data transformation effected by the execution of the path. With such a path we associate the following verification condition:

$$V_\pi : \quad \varphi_i(\vec{y}) \wedge c_\pi(\vec{y}) \quad \rightarrow \quad \varphi_j(f_\pi(\vec{y}))$$

The other type of verification condition is associated with a procedure call. Consider an edge of the following form:



With the (length one) verification path e, we associate the following two verification conditions:

$$V_{in} : \quad \varphi_i(\vec{y}) \wedge c(\vec{y}) \quad \rightarrow \quad p_k(\vec{E}(\vec{y}))$$
$$V_{out} : \quad \varphi_i(\vec{y}) \wedge c(\vec{y}) \wedge q_k(\vec{E}(\vec{y}); \vec{z}') \quad \rightarrow \quad \varphi_j(\vec{y})[\vec{v} \mapsto \vec{z}']$$

where $\varphi_j(\vec{y})[\vec{v} \mapsto \vec{z}']$ is obtained from $\varphi_j(\vec{y})$ by replacing variables in $\vec{v}$ by corresponding variables in $\vec{z}'$.

Soundness of the Method

An assertion network which satisfies all the verification conditions is called an inductive network. An assertion network is defined to be p-invariant if every p-computation σ which reaches location $\ell \in \mathcal{C}$ with data state $\vec{y} = \vec{d}$ satisfies $\vec{d} \models \varphi_\ell$.

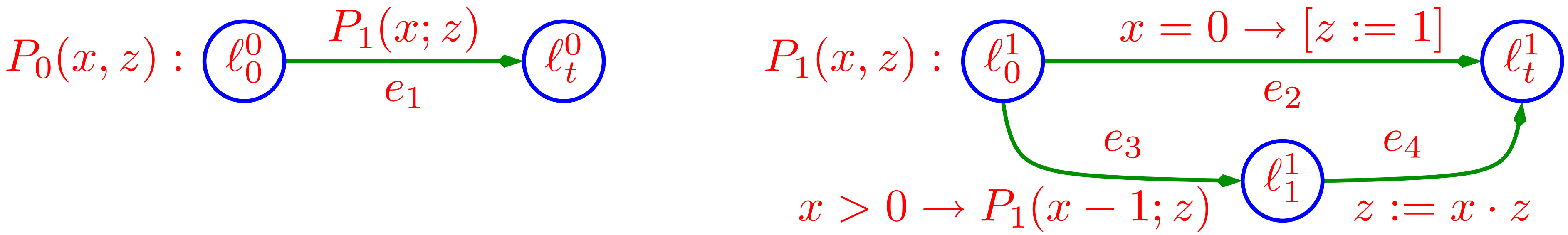
Claim 6. *An inductive assertion network whose assertion at ℓ_0^0 is p_0 is a p_0-invariant network.*

The claim can be proved by induction on the number of cut-points which the computation σ visits.

Corollary 7. *If the network $\mathcal{N}$ is inductive for program P, then P is partially correct w.r.t the specification $\langle p_0, q_0 \rangle$. Furthermore, if $\mathcal{N}$ entails the specification $\langle p, q \rangle$, then P is partially correct w.r.t $\langle p, q \rangle$.*

Example: Factorial

Reconsider the program for computing the factorial of a natural number.



We will prove that this program is partially correct w.r.t the specification

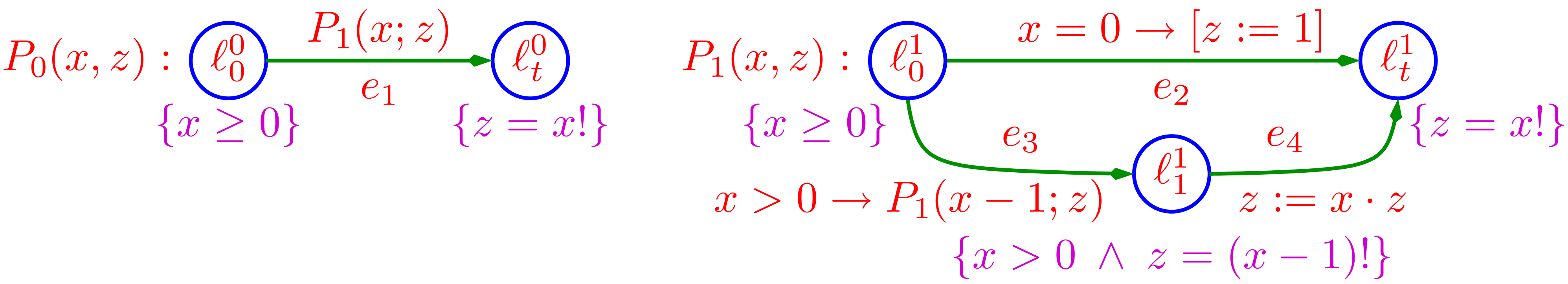
$$p : x \geq 0 \qquad\qquad q : z = x!$$

As the cut-set we take all locations. The proposed assertion network is given by

$$\begin{array}{ll} p_0 = p_1 : & x \geq 0 \\ q_0 = q_1 : & z = x! \\ \varphi_1^1 : & x > 0 \ \wedge \ z = (x-1)! \end{array}$$

The Generated Verification Conditions

The annotated program



gives rise to the following set of valid verification conditions:

$$V_{e_1}^{in}: \quad x \geq 0 \quad \rightarrow \quad x \geq 0$$

$$V_{e_1}^{out}: \quad x \geq 0 \ \wedge \ \underbrace{z' = x!}_{q_1(x,z')} \quad \rightarrow \quad \underbrace{z' = x!}_{q_0[z \mapsto z']}$$

$$V_{e_2}: \quad x \geq 0 \ \wedge \ x = 0 \quad \rightarrow \quad \underbrace{1 = x!}_{q_0(f_{e_1}(x;z))}$$

$$V_{e_3}^{in}: \quad x \geq 0 \ \wedge \ x > 0 \quad \rightarrow \quad \underbrace{x - 1 \geq 0}_{p_1(x-1)}$$

$$V_{e_3}^{out}: \quad x \geq 0 \wedge x > 0 \wedge \underbrace{x > 0 \wedge z' = (x-1)!}_{q_1(x-1,z')} \quad \rightarrow \quad \underbrace{x > 0 \wedge z' = (x-1)!}_{\varphi_1^1(x,z')}$$

$$V_{e_4}: \quad x > 0 \ \wedge \ z = (x-1)! \quad \rightarrow \quad \underbrace{x \cdot z = x!}_{q_1(x,x \cdot z)}$$

Temporal Annotations

We propose to use temporal logic formulas as program annotations. We will list some of the reasons why this may be a good idea.

Often, the programmer wishes to state one or more of the following statements at a control location ℓ:

- On every visit to ℓ, y is non-negative. Can use a conventional assertional annotation $\{x \geq 0\}$.
- I can only reach ℓ if the most recent request has been from customer 3. Can use a past formula annotation $\{(\neg req)\ \mathcal{S}\ (req \wedge customer\_id = 3)\}$.
- Having reached location ℓ, the next response will be to customer 3. Can use a future formula annotation $\{(\neg resp)\ \mathcal{U}\ (resp \wedge customer\_id = 3)\}$.

Temporal Annotations Enable Multiple Reference Points

Traditional assertional annotations enable a reference to a single control point – the one at which the annotation appears. Temporal annotations enable simultaneous reference to multiple control points.

Assume a procedure with input x and output z, which may freely assign values to x and z during execution. Assume that the entry and exit locations are ℓ_{in} and ℓ_{out} respectively. Then

- Partial correctness w.r.t $(\varphi(x), \psi(x,z))$ can be captured by the annotation $\{(\neg \textit{at\_}\ell_{in})\ \mathcal{S}\ (\textit{at\_}\ell_{in} \wedge x = u) \wedge \varphi(u)\ \rightarrow\ \psi(u,z)\}$ at location ℓ_{out}.
- Total correctness w.r.t $(\varphi(x), \psi(x,z))$ can be captured by the annotation $\{\varphi(x)\ \rightarrow\ (\neg \textit{at\_}\ell_{out})\ \mathcal{S}\ (\textit{at\_}\ell_{out} \wedge z = u) \wedge \psi(x,u)\}$ at location ℓ_{in}.
- The fact that variable y decreases between two consecutive visits to location ℓ can be captured by the annotation $\{\textit{at\_}\ell\ \mathcal{S}\ (\neg \textit{at\_}\ell)\ \mathcal{S}\ (\textit{at\_}\ell \wedge y = u)\ \rightarrow\ u > y\}$ at location ℓ.

This last style of temporal annotations has been used in the language TimeC [LPP01] for specifying and implementing real-time constraints by optimizing compilers.

Applications for Run-Time Verification

Temporal annotations will make run-time monitoring more powerful and natural. Already, there are algorithmic approaches to run-time verification of temporal properties of sequential programs.

These algorithms are effective in particular for safety and past properties. But there are some promising approaches also to the treatment of liveness properties. For example, trying to detect whether the validity of a property (e.g. $p\,\mathcal{U}\,q$) has already been determined after observing a finite prefix.

Verifying Temporal Annotations

Floyd's inductive assertion method can be extended to deal with all safety (past-based) properties.

To deal with liveness properties, we have to use well-founded ranking functions, which are the part of Floyd's theory intended to deal with termination and total correctness. Details are still to be worked out.

Temporal Logic Applied to Sequential Programs

For a long time, many researchers held the position that temporal logic has value only in the context of concurrent programs.

With the recent advances in software model checking, automated program analysis and analysis of recursive procedures, it appears that the study of on-going behavior is also useful for the study and development of sequential programs.

Allowing temporal annotations within programs is a very promising approach to the integration of temporal logic with the systematic development of (sequential) programs, and may contribute to meaningful progress in the VSTTE effort.