Reasoning About Programs by
Exploiting the Environment*

Limor Fix
Fred B. Schneider

TR 94-1409
February 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, AFOSR under proposal 93NM312, the National Science Foundation under Grant No. CCR-8701103, and DARPA/NSF Grant No. CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies. Limor Fix is also supported, in part, by a Fulbright post-doctoral award.
Reasoning about Programs by
Exploiting the Environment

Limor Fix
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

February 2, 1994

ABSTRACT
A method for making aspects of a computational model explicit in the formulas of a programming logic is given. The method is based on a new notion of environment—an environment augments the state transitions defined by a program's atomic actions rather than being interleaved with them. Two simple semantic principles are presented for extending a programming logic in order to reason about executions feasible in various environments. The approach is illustrated by (i) discussing a new way to reason in TLA and Hoare-style programming logics about real-time and by (ii) deriving the first TLA and Hoare-style proof rules for reasoning about schedulers.

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, AFOSR under proposal 93NM312, the National Science Foundation under Grant No. CCR-8701103, and DARPA/NSF Grant No. CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies. Limor Fix is also supported, in part, by a Fulbright postdoctoral award.
1. Introduction

What behaviors of a concurrent program are possible may depend on the scheduler, instruction timings, and other aspects of the environment in which that program executes. For example, consider the program of Figure 1.1. Process $P_1$ executes an atomic action that sets $y$ to 1 followed by one that sets $y$ to 2. Concurrently, process $P_2$ executes an atomic action that sets $y$ to 3. If all behaviors of this concurrent program were possible, then the final value of $y$ would be 2 or 3. The environment, however, may rule out certain behaviors.

- Suppose $P_1$ has higher-priority than $P_2$ and the environment selects between executable atomic actions by using a priority scheduler. Behaviors in which actions of $P_2$ execute before those of $P_1$ are now infeasible, and the final value of $y$ cannot be 2.

- Suppose the environment uses a first-come first-served scheduler to select between executable atomic actions. Behaviors in which $P_2$ executes after the second action of $P_1$ are now infeasible, and the final value of $y$ cannot be 3.

Thus, changing the environment can affect what properties a program satisfies.

Programming logics usually axiomatize program behavior under certain assumptions about the environment. Logics to reason about real-time, for example, axiomatize assumptions about how time advances while the program executes. These assumptions abstract the effects of the scheduler and the execution times of various atomic actions. A logic to reason about the consequences of resource constraints would similarly have to axiomatize assumptions about resource availability.

If assumptions about an environment are made when defining a programming logic, then changes to the environment may require changes to the logic. Previously feasible behaviors could become infeasible when the assumptions are strengthened; a logic for the original environment would then be incomplete for this new environment. Weakening the assumptions could add feasible behaviors; the logic for the original environment would then become unsound. For example, any of the programming logics for shared-memory concurrency (e.g. [OG76]) could be used to prove that program of Figure 1.1 terminates with $y = 2$ or $y = 3$. But, these logics must be changed to prove that $y = 2$ necessarily holds if a first-come first served scheduler is being used or that $y = 3$ necessarily holds if a priority scheduler is used. As another example, termination of the program in Figure 1.2 depends on whether unfair behaviors are feasible. (Usually they are not.) Logics, like the temporal logic of [MP89], that assume a fair scheduler become unsound when this assumption about the environment is relaxed.

```
cobegin
    P_1: y := 1; y := 2
    //
    P_2: y := 3
coend
```

Figure 1.1. A concurrent program
cobegin
  $P_1$: $b := \text{false}$
  //
  $P_2$: $b \rightarrow \text{skip}$ od
coend

Figure 1.2. Termination with fairness

This paper explores the design of programming logics in which assumptions about the environment can be given explicitly. Such logics allow us to prove that all feasible behaviors of a program satisfy a property, where the characterization of what is feasible is now explicit and subject to change. We give two semantic principles—program reduction and property reduction—for extending a programming logic so that explicit assumptions about an environment can be exploited in reasoning. These principles allow extant programming logics to be extended for reasoning about the effects of various fairness conditions, schedulers, and models of real-time; a new logic need not be defined every time a new model of computation is postulated. We illustrate the application of our two principles using TLA [L91] and a Hoare-style Proof Outline Logic [S94]. In TLA, programs and properties are both represented using a single language; in Proof Outline Logic these two languages are distinct.

The remainder of this paper is structured as follows. In section 2, our program and property reduction principles are derived. Then, in section 3, program reduction is applied to TLA. In section 4, property reduction is used to drive an extension to a Hoare-style logic. Section 5 puts this work in context. The appendix contains the completeness proof for the extended Hoare-style logic.

2. Formalizing and Exploiting the Environment

A programming logic comprises a sound and complete deductive system for verifying that a given program satisfies a property of interest. We write $(S, \Psi) \in \text{Sat}$ to denote that a program $S$ satisfies a property $\Psi$; each programming logic will have its own syntax for saying this. In any given programming logic, a program language is used to specify $S$ and a property language, perhaps identical to the program language, is used to specify $\Psi$.

Usually, both the program $S$ and the property $\Psi$ define sets of behaviors, where a behavior is a mathematical object that encodes a sequence of state transitions resulting from program execution, and a state is a mapping from variables to values. Notice that

- the set $\llbracket S \rrbracket$ of behaviors for a program $S$ constrains only the values of program variables\(^1\), and
- the set $\llbracket \Psi \rrbracket$ of behaviors for a property $\Psi$ may also constrain the values of variables that are not program variables.

A program $S$ satisfies a property $\Psi$ exactly when all of the behaviors of $S$ are behaviors permitted by $\Psi$:

\(^1\)Program variables include all those declared explicitly in the program as well as others, like program counters and message buffers, concerning aspects of the state implicitly involved in executing the program.
\langle S, \Psi \rangle \in \text{Sat} \quad \text{if and only if} \quad [S] \subseteq [\Psi]

(2.1)

The environment in which a program executes defines a property too. This property contains any behavior that is not precluded by one or another aspect of the environment. For example, a priority scheduler precludes behaviors in which atomic actions from low-priority processes are executed instead of those from high-priority processes. As another example, the environment might define the way a distinguished variable \textit{time} (say) changes in successive states, taking into account the processor speed for each type of atomic action.

For \( E \) the property defined by the environment, the \textit{feasible behaviors} of a program \( S \) under \( E \) are those behaviors of \( S \) that are also in \( E: [S] \cap [E] \). A program \( S \) satisfies a property \( \Psi \) under an environment \( E \), denoted \( \langle S, E, \Psi \rangle \in \text{ESat} \), if and only if every feasible behavior of \( S \) under \( E \) is in \( \Psi \):

\[ \langle S, E, \Psi \rangle \in \text{ESat} \quad \text{if and only if} \quad ([S] \cap [E]) \subseteq [\Psi] \]

(2.2)

Thus, a sound and complete deductive system for verifying \( \langle S, E, \Psi \rangle \in \text{ESat} \) would permit us to prove properties of programs under various assumptions about schedulers, execution times, and so on.

Defining a separate logic to prove \( \langle S, E, \Psi \rangle \in \text{ESat} \) is not always necessary if a logic to prove \( \langle S, \Psi \rangle \in \text{Sat} \) is available. For properties \( \Phi \) and \( \Psi \), let property \( \Phi \cap \Psi \) be \( [\Phi] \cap [\Psi] \) and let property \( \Phi \cup \Psi \) be \( [\Phi] \cup [\Psi] \). Then, one reduction from \text{ESat} to \text{Sat} is derived as follows.

\[ \langle S, E, \Psi \rangle \in \text{ESat} \]
\[ \text{iff} \quad \text{definition (2.2) of ESat} \]
\[ ([S] \cap [E]) \subseteq [\Psi] \]
\[ \text{iff} \quad \text{definition (2.1) of Sat} \]
\[ \langle S \cap E, \Psi \rangle \in \text{Sat} \]

Thus we have:

**Program Reduction:** \( \langle S, E, \Psi \rangle \in \text{ESat} \) if and only if \( \langle S \cap E, \Psi \rangle \in \text{Sat} \).

(2.3)

Program Reduction is useful if the logic for \( \langle S, \Psi \rangle \in \text{Sat} \) has a program language that is closed under intersection with the language used to define environments. Section 3 shows this to be the case for Lamport's TLA; it is also the case for most other temporal logics.

A second reduction from \text{ESat} to \text{Sat} is based on using the environment to modify the property (rather than the program).

\[ \langle S, E, \Psi \rangle \in \text{ESat} \]
\[ \text{iff} \quad \text{definition (2.2) of ESat} \]
\[ ([S] \cap [E]) \subseteq [\Psi] \]
\[ \text{iff} \quad \text{set theory} \]
\[ [S] \subseteq ([\Psi] \cup [E]) \]
\[ \text{iff} \quad \text{definition (2.1) of Sat} \]
\[ \langle S, \Psi \cup E \rangle \in \text{Sat} \]

This proves:

**Property Reduction:** \( \langle S, E, \Psi \rangle \in \text{ESat} \) if and only if \( \langle S, \Psi \cup E \rangle \in \text{Sat} \).

(2.4)

Property reduction imposes no requirement on the program language, but does require that the property language be closed under union with the complement of properties that might be defined by environments. An example of a logic whose property language satisfies this closure condition is
CTL* [EH86].

When neither reduction principle applies, then we can reason about the effects of an environment by extending the logic being used to establish \((S, \forall) \in \text{Sat}\). Extensions to the program language allow Program Reduction to be applied; extensions to the property language allow Property Reduction to be applied. Section 4 illustrates how this might be done, by extending the property language of a Hoare-style logic called Proof Outline Logic.

3. Environments for TLA

The Temporal Logic of Actions (TLA) is a linear-time temporal logic in which programs and properties are represented as formulas. Thus, the program language and property language of TLA are one and the same. This single language includes the usual propositional connectives, and the TLA formula \(F \land G\) defines a property that is the intersection of the properties defined by \(F\) and \(G\). TLA is, therefore, an ideal candidate for Program Reduction.

3.1. TLA Overview

A TLA state predicate is a predicate logic formula over some variables.\(^2\) The usual meaning is ascribed to \(s \models p\) for a state \(s\) and a state predicate \(p\): when each variable \(v\) in \(p\) is replaced by its value \(s(v)\) in state \(s\), the resulting formula is equivalent to \(true\). For example, in a state \(s\) that maps \(y\) to 14 and \(z\) to 22, \(s \models y + 1 < z\) holds because \(s(y) + 1 < s(z)\) equals \(14 + 1 < 22\), which is equivalent to \(true\).

A TLA action is a predicate logic formula over unprimed variables and primed variables. Actions are interpreted over pairs of states. The unprimed variables are evaluated in the first state \(s\) of the pair \((s, t)\) and the primed variables are evaluated, as if unprimed, in the second state \(t\) of the pair. For example, if \(s(y)\) equals 13 and \(t(y)\) equals 16 then \((s, t) \models y + 1 < y'\) holds because \(s(y) + 1 < t(y)\) is equal to \(13 + 1 < 16\), or, \(true\).

In order to facilitate writing actions that are invariant under stuttering, TLA provides an abbreviation. For action \(\mathcal{A}\) and list \(\bar{x}\) of variables \(x_1, x_2, ..., x_n\), the action\(^3\) \([\mathcal{A}]_{\bar{x}}\) is satisfied by any pair \((s, t)\) of states such that \((s, t) \models \mathcal{A}\) or the values of the \(x_i\) are unchanged between \(s\) and \(t\). Writing \(\bar{x}'\) to denote the result of priming every variable in \(\bar{x}\), we get:

\[[\mathcal{A}]_{\bar{x}}: \; \mathcal{A} \lor \bar{x} = \bar{x}'\]

TLA actions define state transitions. Therefore, they can be used to describe the next-state relation of a concurrent program, a single sequential process, or any piece thereof. For this purpose, it is useful to define a state predicate satisfied by any state from which transition is possible due to an action \(\mathcal{A}\). That state predicate, \(\text{Enbl}(\mathcal{A})\), is defined by:

\(s \models \text{Enbl}(\mathcal{A})\) if and only if \(\exists t: \; (s, t) \models \mathcal{A}\)

Each formula \(\Phi\) of TLA defines a property \(\llbracket \Phi \rrbracket\), which is the set of behaviors that satisfy \(\Phi\), where a behavior is represented by a sequence of states. Let \(\sigma\) be a behavior \(s_0, s_1, ..., s_n\), let \(p\) be a state predicate, let \(\mathcal{A}\) be an action, and let \(\bar{x}\) be a list of variables. The syntax of the elementary formulas of

\(^2\)We assume that variable names do not contain the character """" (prime).

\(^3\)TLA actually allows subscript \(\bar{x}\) to be an arbitrary state function whose value will remain unchanged.
TLA, along with the property defined by each, is:

\[
\begin{align*}
\sigma &\in \llbracket p \rrbracket \quad \text{iff} \quad s_0 \models p \\
\sigma &\in \llbracket \Box [A] \rrbracket \quad \text{iff} \quad \text{For all } i, i \geq 0: (s_i, s_{i+1}) \models [A] \overline{x}
\end{align*}
\]

The remaining formulas of TLA are formed from these, as follows. Let \( \Phi \) and \( \Psi \) be elementary TLA formulas or arbitrary TLA formulas.

\[
\begin{align*}
\sigma &\in \llbracket \neg \Phi \rrbracket \quad \text{iff} \quad \sigma \not\in \llbracket \Phi \rrbracket \\
\sigma &\in \llbracket \Phi \lor \Psi \rrbracket \quad \text{iff} \quad \sigma \in \llbracket \Phi \rrbracket \lor \llbracket \Psi \rrbracket \\
\sigma &\in \llbracket \Phi \land \Psi \rrbracket \quad \text{iff} \quad \sigma \in \llbracket \Phi \rrbracket \land \llbracket \Psi \rrbracket \\
\sigma &\in \llbracket \Phi \Rightarrow \Psi \rrbracket \quad \text{iff} \quad \sigma \in \llbracket \neg \Phi \lor \Psi \rrbracket \\
\sigma &\in \llbracket \Box \Phi \rrbracket \quad \text{iff} \quad \text{For all } i, i \geq 0: (s_i, s_{i+1}) \models \Phi \\
\sigma &\in \llbracket \Diamond \Phi \rrbracket \quad \text{iff} \quad \sigma \in \llbracket \neg \Box \neg \Phi \rrbracket
\end{align*}
\]

A TLA formula \( \Phi \) is \textit{valid} if and only if for every behavior \( \sigma, \sigma \in \llbracket \Phi \rrbracket \) holds. Validity of \( \Phi \Rightarrow \Psi \) implies that every behavior \( \sigma \) is in \( \llbracket \Phi \Rightarrow \Psi \rrbracket \). From the definition above for \( \sigma \in \llbracket \Phi \Rightarrow \Psi \rrbracket \), we have that if \( \Phi \Rightarrow \Psi \) is valid then every \( \sigma \) in \( \llbracket \Phi \rrbracket \) is also in \( \llbracket \Psi \rrbracket \). Accordingly, we conclude:

\[
\Phi \Rightarrow \Psi \text{ is valid if and only if } \langle \Phi, \Psi \rangle \in \text{Sat}
\]

(3.1)

To prove that a program \( S \) satisfies a property \( \Psi \) using TLA, we

1. construct a TLA formula \( \Phi_S \) such that \( \llbracket \Phi_S \rrbracket \) is the set of behaviors of \( S \), and
2. prove \( \Phi_S \Rightarrow \Psi \) valid.

As an example, we return to the program of §1. It is reproduced in Figure 3.1, with each atomic action labeled. The TLA formula \( \Phi_S \) that characterizes behaviors for this program is

\[
\Phi_S: \text{Init}_S \land \Box [A_S]y, pc_1, pc_2
\]

where \( \text{Init}_S \) is a state predicate defining initial states of the program's behavior and \( A_S \) is a TLA action that characterizes the program's next-state relation. In defining the effect of each atomic action, variable \( pc_i \) denotes the program counter for process \( P_i \) and value "\( \downarrow \)" is assumed to be different from the entry (control) point for any atomic action of the program.

cobegin
\[
\begin{align*}
P_1: &\quad \alpha: \ y := 1 \\
&\quad \beta: \ y := 2 \\
&\quad //
\end{align*}
\]

\[
P_2: \quad \gamma: \ y := 3
\]

doend

Figure 3.1. A concurrent program
Initₜ: \( pc_1 = \alpha \land pc_2 = \gamma \)

\( S \): \( \alpha \lor \beta \lor \gamma \)

\( \alpha \): \( pc_1 = \alpha \land pc_1' = \beta \land y' = 1 \land pc_2 = pc_2' \)

\( \beta \): \( pc_1 = \beta \land pc_1' = \downarrow \land y' = 2 \land pc_2 = pc_2' \)

\( \gamma \): \( pc_2 = \gamma \land pc_2' = \downarrow \land y' = 3 \land pc_1 = pc_1' \)

3.2. Exploiting an Environment with TLA

If the property defined by an environment can be characterized in TLA, then Program Reduction can be used to reason about feasible behaviors under that environment. We prove \( \Phi \land E \Rightarrow \Psi \) to establish that behaviors of the program characterized by \( \Phi \) under the environment characterized by \( E \) are in the property characterized by \( \Psi \):\

\[ \Phi \land E \Rightarrow \Psi \text{ is valid} \]

iff «definition (3.1)»

\[ \langle \Phi \land E, \Psi \rangle \in \text{Sat} \]

iff «definition (2.1)»

\[ \llbracket \Phi \land E \rrbracket \subseteq \llbracket \Psi \rrbracket \]

iff «definition (2.1)»

\[ \llbracket F \land G \rrbracket = \llbracket F \rrbracket \cap \llbracket G \rrbracket \]

iff «Program Reduction (2.3)»

\[ \langle \Phi \land E, \Psi \rangle \in \text{ESat} \]

The utility of this method depends on (i) being able to prove \( \Phi \land E \Rightarrow \Psi \) when it is valid and (ii) being able to characterize in TLA those aspects of environments that interest us. A complete deductive system for TLA (see [L91], for example) will, by definition, be complete for proving \( \Phi \land E \Rightarrow \Psi \). In fact, this is one of the advantages of using Program Reduction to extend a complete proof system for Sat into a proof system for ESat—the complete proof system for ESat comes at no cost. Examples in the remainder of this section convey a sense for how an environment is represented by a TLA formula.

3.3. Schedulers as TLA formulas

If there are more processes than processors in a computer system, then processors must be shared. This sharing is usually implemented by the scheduler of an operating system. To use Program Reduction with TLA and reason about execution of a program under a given scheduler, we write a TLA formula \( E \) to characterize that scheduler.

Many schedulers implement safety properties—they rule out certain assignments of processors to processes. Formalizations for these schedulers have much in common. Let \( \Pi \) be the set of processes to be executed in a system with \( N \) processors. For each process \( \pi \), two pieces of information are maintained (in some form) by a scheduler:

- \( \text{active}_\pi \): whether there is a processor currently allocated to \( \pi \)
- \( \text{rank}_\pi \): a value used to determine whether a processor should be allocated to \( \pi \)
Only a single atomic action from one process can be executed at any time by a processor. This restriction is formalized as predicate $Alloc(N)$, which bounds the number of processes to which $N$ processors can be allocated at any time:\footnote{We use the notation ($\#x \in P: R$) for "the number of distinct values of $x$ in $P$ for which $R$ holds".}

$$Alloc(N): \ (#\pi \in \Pi: active_{\pi}) \leq N$$

The restriction that processes that have processors allocated are the only ones that advance is formalized in terms of $A_{\pi}$, the next-state relation for a process $\pi$. We assume that these next-state relations are disjoint.

$$Pgrs(\pi): \ A_{\pi} \Rightarrow active_{\pi}$$

Finally, we formalize as $Run(\pi)$ the requirement that $active_{\pi}$ holds only for those processes with sufficiently large rank.

$$Run(\pi): \ active_{\pi} \Rightarrow |larger(\pi)| < N$$

where:

$$larger(\pi): \ \{\pi' | rank_{\pi} < rank_{\pi'}\}$$

In a fixed-priority scheduler, there is a fixed value $v_{\pi}$ associated with each process $\pi$. A process that has not terminated and has higher priority is executed in preference to a process having a lower priority. This is ensured by assigning ranks as follows.

$$Prio(\pi): \ (pc_{\pi} \neq \downarrow \Rightarrow (rank_{\pi} = v_{\pi})) \land (pc_{\pi} = \downarrow \Rightarrow (rank_{\pi} = 0))$$

A fixed-priority scheduler is thus characterized by

$$FixedPrio: -\square[Alloc(N) \land (\forall \pi \in \Pi: Pgrs(\pi) \land Run(\pi) \land Prio(\pi))]$$

where $\bar{x}$ is a list of all the variables in the system. For example, $\bar{x}$ for the program of Figure 3.1 would have $pc_{1}, pc_{2}, y, active_{p_{1}}, rank_{p_{1}}, active_{p_{2}},$ and $rank_{p_{2}}$.

In a first-come first-served scheduler, processes are ranked in accordance with elapsed time since last executed. We can model this by assigning ranks that are increased for processes that have not had an action executed.

$$Age(\Pi): \ (\forall \pi \in \Pi: (A_{\pi} \Rightarrow (rank'_{\pi} = 0)) \land (\neg A_{\pi} \Rightarrow (rank'_{\pi} = rank_{\pi} + 1)))$$

A first-come, first-served scheduler is therefore characterized by

$$FCFS: \ (\forall \pi \in \Pi: rank_{\pi} = 0) \land -\square[Alloc(N) \land (\forall \pi \in \Pi: Pgrs(\pi) \land Run(\pi)) \land Age(\Pi)]$$

where $\bar{x}$ is a list of all the variables in the system.

Both of these schedulers can allocate a processor to a process, even though that process may be unable to make progress. It is wasteful to allocate a processor to process $\pi$ when $Enbl(A_{\pi})$ does not hold (because $\pi$ has terminated or because its next atomic action is not enabled). A variant of FixedPrio that allocates processors only to non-terminated and enabled higher-priority processes is:

$$EnblFixedPrio: -\square[Alloc(N) \land (\forall \pi \in \Pi: Pgrs(\pi) \land Run(\pi) \land EnblPrio(\pi))]$$

where
\[ \text{EnblPrio}(\pi): (\text{Enbl}(A_\pi) \Rightarrow (\text{rank}_\pi = v_\pi)) \land (\neg \text{Enbl}(A_\pi) \Rightarrow (\text{rank}_\pi = 0)) \]

As before, \( x \) is a list of all the variables in the system.

A difficulty with assigning fixed priorities to processes is that execution of a high-priority process can be delayed awaiting progress by processes with lower-priorities. For example, suppose a high-priority process \( \pi_H \) is awaiting some lock to be freed, so \( \pi_H \) is not enabled. If that lock is owned by a lower-priority process \( \pi_L \), then execution of \( \pi_H \) cannot proceed until \( \pi_L \) executes. This is known as a priority inversion \( [\text{SRL90}][\text{BMS93}] \), because execution of a high-priority process depends on resources being allocated to a lower-priority process.

Priority Inheritance schedulers give preference to low-priority processes that are blocking high-priority processes. This is done by changing process priorities. The low-priority process inherits a new, higher priority from any higher-priority process it blocks. Priority inheritance schedulers exhibit improved worst-case response times in systems of tasks \( [\text{SRL90}] \), and they have become important in the design of real-time systems.

A priority inheritance scheduler must know what processes are blocked and how to unblock them. In systems where acquiring a lock is the only operation that blocks a process, deducing this information is easy: execution of the process that has acquired a lock is the only way that a process awaiting that lock becomes unblocked.

To describe systems with locks in TLA, we employ a variable \( lock_i \) for each lock; TLA actions for acquiring and releasing a lock by process \( \pi \) are:

\[
\begin{align*}
\text{acquire}(lock_i, \pi) & : \text{lock}_i = \text{FREE} \land \text{lock}_i' = \pi \\
\text{release}(lock_i) & : \text{lock}_i' = \text{FREE}
\end{align*}
\]

Notice that \( lock_i = \text{FREE} \) is implied by \( \text{Enbl}(A_\pi) \) when process \( \pi \) is waiting to acquire \( lock_i \).

In a priority inheritance scheduler, each process \( \pi \) is assumed to have a priority \( v_\pi \). The rank of a process \( \pi \) is the maximum of \( v_\pi \) and the priorities assigned to processes that are blocked by \( \pi \). Thus, \( \text{rank}_\pi \) is the maximum of \( v_p \) for the process \( p \) satisfying \( lock_i = p \) (i.e., the priority of the current lock holder) and \( v_q \) for the process \( q \) satisfying \( \text{Enbl}(p) \Rightarrow (lock_i = \text{FREE}) \) (i.e., the priority of the process attempting to acquire \( lock_i \)). For simplicity, we assume a system having a single lock, \( lock \).

\[
\begin{align*}
\text{Priolnher}(\pi) & : [(\neg \text{Enbl}(A_\pi) \Rightarrow (\text{rank}_\pi = 0)) \land \\
& (lock = \pi \land \text{Enbl}(A_\pi)) \Rightarrow (\text{rank}_\pi = (\max p \in \Pi: (\text{Enbl}(p) \Rightarrow lock = \text{FREE}) \lor lock = p \Rightarrow v_p))) \land \\
& (lock \neq \pi \land \text{Enbl}(A_\pi) \Rightarrow (\text{rank}_\pi = v_\pi))_{\bar{x}}
\end{align*}
\]

Again, \( \bar{x} \) is a list of all the variables in the system. A priority inheritance scheduler is thus characterized by

\[
\text{InhPrio} : \Box[\text{Alloc}(N) \land (\forall \pi \in \Pi : Pgrs(\pi) \land Run(\pi) \land \text{Priolnher}(\pi))]_{\bar{x}}
\]

Not all schedulers are safety properties. Even schedulers that implement safety properties are often abstracted in programming logics as implementing (weaker) liveness properties. Such a liveness property gives conditions under which an action or process will be executed eventually. A simple example is the following, which implies that an enabled process with sufficiently high priority will execute.

\[
\text{FAIR} : (\forall \pi \in \Pi : \Diamond \Box (\pi \in \text{TOP}(n, \Pi) \land \text{Enbl}(\pi)) : \neg \Diamond \Box (\neg A_\pi)_{\bar{x}})
\]
Other examples of such liveness properties include weak fairness $WF_x(\mathcal{A})$ and strong fairness $SF_x(\mathcal{A})$ of TLA.

### 3.4. Real time in TLA

The correlation between execution of a program and the advancement of time is largely an artifact of the environment in which that program executes. The scheduler, the number of processors, and the availability of other resources all play a role in determining when a process may take a step. To reason with TLA about properties satisfied by a program in such an environment, we simply characterize the way time advances and then use Program Reduction. Various models of real-time one finds in the literature differ only in their characterization of how time advances.

When only a single processor is assumed, then process execution is interleaved on that processor. One way to abstract this is to associate two constants with each atomic action $\alpha$:

- $e_\alpha$: the fixed execution time of atomic action $\alpha$ on a bare machine

---

5 $A'$ denotes the formula obtained by priming each un-primed free variable in $A$.

6 The choice of $B$ is based on applying the Temporal Logic axiom $(\Box E \land \Box F) = \Box (E \land F)$. 
\( \delta_\alpha \): the maximum time that can elapse from the time that the processor is allocated for execution of \( \alpha \) until \( \alpha \) starts executing.

Execution of \( \alpha \) is thus correlated with the passage of between \( e_\alpha \) and \( e_\alpha + \delta_\alpha \) time units.

The following TLA formula is satisfied by such behaviors. Variable \( T \) is the current time and \( \text{ATOM}(S) \) is the set of atomic actions in \( S \). Recall that \( A_\alpha \) defines atomic action \( \alpha \).

\[
T=0 \land \square[ \bigwedge_{\alpha \in \text{ATOM}(S)} (A_\alpha \Rightarrow (T+e_\alpha \leq T' \leq T + e_\alpha + \delta_\alpha))]_x
\]

As before, \( x \) is a list of all variables in the system.

Another common model of how time advances abstracts the case where each process is executed on its own processor. We assume that the next action to be executed at process \( \pi \) is uniquely defined at each control point. (Other assumptions are possible, and these can be formalized also.) We formalize this environment in TLA, by using a separate variable \( T_\pi \) for each process \( \pi \):

\( T_\pi \): the time process \( \pi \) arrived at its current state.

System time \( T \) is the maximum of the \( T_\pi \):

\[
\text{SysTme} : \quad T = \max_{\pi \in \Pi} (T_\pi)
\]

And each individual process \( \pi \) must execute its next action \( \alpha \) (say) before \( e_\alpha + \delta_\alpha \) has elapsed from the time \( \pi \) reached its current state. Let the label on action \( \alpha \) be "\( \alpha \)".

\( \text{LclTme} : \quad (\forall \pi \in \Pi : \text{pc}_\pi = \alpha : T - T_\pi \leq e_\alpha + \delta_\alpha) \)

The range \( \text{pc}_\pi = \alpha \) is satisfied by states in which the program counter for process \( \pi \) indicates that \( \alpha \) is the next atomic action to be executed; the body requires \( \alpha \) to be executed before the system’s time has advanced too far.

Finally, the value of \( T_\pi \) changes iff an atomic action from process \( \pi \) is executed:

\( \text{LclTmeUpdt} : \quad (\forall \pi \in \Pi : (\forall \alpha \in \text{ATOM}(S) : A_\alpha : T_\pi + e_\alpha \leq T'_\pi \leq T_\pi + e_\alpha + \delta_\alpha \\
\quad \land (\forall \phi \in \Pi : \phi \neq \pi : T'_\phi = T_\phi)) \)

Here, the range is satisfied only by steps attributed to atomic action \( \alpha \) of process \( \pi \); the body causes all of the \( T_\pi \) to be updated.

Putting all these together, we get a TLA formula characterizing this model of real time:

\[
T=0 \land (\forall \pi \in \Pi : T_\pi = 0) \land \square[ \bigwedge_{\alpha \in \text{ATOM}(S)} (\text{SysTme} \land \text{LclTme} \land \text{LclTmeUpdt})]_x
\]

(3.4)

**An Old-fashioned Recipe**

The scheme just described works by restricting the transitions allowed by each action. These restrictions ensure that an action only executes when its starting and ending times are as prescribed by the real-time model. Thus, the approach regards the environment as augmenting each action of the original system. The environment executes simultaneously with the system's actions.

A somewhat different approach to reasoning about real-time with TLA is described by Lamport and Abadi in "An old-fashioned recipe for real-time" [AL91]. That recipe is extended for handling schedulers in [LJJ93]. Like our scheme, the recipe does not require changes to the language or deductive system of TLA. However, unlike our scheme, additional actions are used to handle the passage of time. These new actions interleave with the original program actions, updating a clock.
and some count-down timers.

There seems to be no technical reason to prefer one approach to the other. In the examples we have checked, the old-fashioned recipe is a bit cumbersome. A variable now analogous to our variable T is used to keep track of the current time, and a variable, called a timer, is associated with each atomic action whose execution timing is constrained. Timers ensure (i) that the new actions to advance now are disabled when actions of the original program must progress and (ii) that actions of the original program are disabled when now has not advanced sufficiently. The timers, now, and added actions implement what amounts to a discrete-event simulation that causes time to advance and actions to be executed in an order consistent with timing constraints. To write real-time specifications, it suffices to learn the few TLA idioms in [AL91] and repeat them. However, to prove properties from these specifications, the details of this discrete event simulation must be mastered.

4. Environments for a Hoare-style Proof Outline Logic

We now turn our attention to a second programming logic—one that is quite different in character from TLA. The formulas of a Hoare-style logic are imperative programs in which an assertion is associated with each control point. This rules out Program Reduction (2.3), because imperative programming languages are generally not closed under intersection of any sort. Similarly, Property Reduction (2.4) is ruled out because the property language, annotated program texts, also lacks the necessary closure. However, it is not difficult to extend the property language of a Hoare-style logic and then apply Property Reduction (2.4). An example of such an extension is given in this section.

4.1. A Hoare-style Logic

Consider a simple programming language having assignment, sequential composition, and parallel composition statements. An example program is given in Figure 4.1; it is equivalent to the program of Figure 1.1.

The syntax of programs in our language is given by the following grammar. There, \( \lambda \) is a label, \( x \) is a program variable, and \( E \) is an expression over the program variables.

\[
S ::= \lambda: [x := E] \ | \ \lambda: [S ; S] \ | \ \lambda: [S // S]
\]

Every label in a program is assumed to be unique. In the discussion that follows, the label on the entire program is used to name that program. In addition, for a statement \( \lambda: [\ldots] \), we call "\( \lambda: [\ldots] \) the

\[
\begin{align*}
\lambda : [ & \lambda_1 : [ \lambda_{11} : [y := 1]; \\
& \lambda_{12} : [y := 2]]\\
& // \\
& \lambda_2 : [y := 3]]
\end{align*}
\]

Figure 4.1. Simple Program

---

7Constraint-maintenance languages are the obvious exception.

8Handling an imperative language with if and do is not fundamentally different.
opening of λ, call "]" the closing, and define \( \text{Lab}(\lambda) \) to be the set containing label \( \lambda \) and all labels used between the opening and closing of \( \lambda \).

A program state assigns values to the program variables and to control variables. The control variables for a program \( \lambda \) are \( \text{at}(\lambda') \), \( \text{in}(\lambda') \), and \( \text{after}(\lambda') \) for every label \( \lambda' \) in \( \text{Lab}(\lambda) \).

The set \( \Sigma \) of program states contains only those states satisfying certain constraints on the values of control variables. These constraints are given in Figure 4.2. They ensure that the control variables encode plausible values of program counters. For example, the constraints rule out the possibility that control variables \( \text{at}(\lambda) \) and \( \text{after}(\lambda) \) are both \textit{true} in a state. As another example, the constraints imply that any state for program \( \lambda \) of Figure 4.1 assigning \textit{true} to \( \text{after}(\lambda_{11}) \) must also assign \textit{true} to \( \text{at}(\lambda_{12}) \).

The executions of a program \( \lambda \) defines a set of behaviors. It will be convenient to represent a behavior using a triple \( \langle \sigma, i, j \rangle \), where \( \sigma \) is an infinite sequence\(^9\) of states, \( i \) is natural number, and \( j \) is a natural number satisfying \( i \leq j \) or \( i = \infty \). Informally, behavior \( \langle \sigma, i, j \rangle \) models a (possibly partial) execution starting in state \( \sigma[i] \) that produces sequence of states \( \sigma[i..j] \). Prefix \( \sigma[..i-1] \) is the sequence of states that precedes the execution; suffix \( \sigma[j..] \) models subsequent execution.

Each state \( s \) of a program \( \lambda \) satisfies:

C0: \( s \vdash (\text{in}(\lambda) \neq \text{after}(\lambda)) \)

C1: \( s \vdash \neg (\text{at}(\lambda) \land \text{after}(\lambda)) \)

C2: \( s \vdash (\text{at}(\lambda) \Rightarrow \text{in}(\lambda)) \)

C3: For every assignment statement \( \lambda: [x := E] \):
   \( s \vdash (\text{at}(\lambda) = \text{in}(\lambda)) \)

C4: For every sequential composition \( \lambda: [\lambda_1: [S \ 1]; \ \lambda_2: [S \ 2]] \):
   \[ s \vdash (\text{at}(\lambda) = \text{at}(\lambda_1)) \]
   \[ s \vdash (\text{after}(\lambda) = \text{after}(\lambda_2)) \]
   \[ s \vdash (\text{at}(\lambda_1) = \text{at}(\lambda_2)) \]
   \[ s \vdash ((\text{in}(\lambda_1) \lor \text{in}(\lambda_2)) \Rightarrow \text{in}(\lambda)) \]
   \[ s \vdash \neg (\text{in}(\lambda_1) \land \text{in}(\lambda_2)) \]

C5: For every parallel composition \( \lambda: [\lambda_1: [S \ 1] // \lambda_2: [S \ 2]] \):
   \[ s \vdash (\text{at}(\lambda) = (\text{at}(\lambda_1) \land \text{at}(\lambda_2))) \]
   \[ s \vdash (\text{after}(\lambda) = (\text{after}(\lambda_1) \land \text{after}(\lambda_2))) \]
   \[ s \vdash (\text{in}(\lambda) = ((\text{in}(\lambda_1) \lor \text{after}(\lambda_1)) \land (\text{in}(\lambda_2) \lor \text{after}(\lambda_2)) \land \neg (\text{after}(\lambda_1) \land \text{after}(\lambda_2)))) \]

Figure 4.2. Constraints on control variables

---

\(^9\)For an infinite sequence \( \sigma = s_0 s_1 \ldots \) we write: \( \sigma[i] \) to denote \( s_i \); \( \sigma[..i] \) to denote prefix \( s_0 s_1 \ldots s_i \); \( \sigma[i..] \) to denote suffix \( s_i s_{i+1} \ldots \); and \( \sigma[i..j] \), where \( i \leq j \), to denote subsequence \( s_i \ldots s_j \).
Formally, we define the set \( \llbracket \lambda \rrbracket \) of behaviors for a program \( \lambda \) in terms of relations \( R_{\lambda'}: [x \Rightarrow E] \) for the assignments \( \lambda' \) in \( \lambda \):

\[
(s, t) \in R_{\lambda': [x \Rightarrow E]} \quad \text{iff} \quad s = at(\lambda'), \ t = after(\lambda'), \ t(x) = s(E), \ \text{and} \ s(v) = t(v) \text{ for all program variables } v \text{ different from } x.
\]  

(4.1)

Let \( Assig(\lambda) \) be the subset of \( Lab(\lambda) \) that are labels on assignment statements in \( \lambda \). Behavior \( \langle \sigma, i, j \rangle \) is defined to be an element of \( \llbracket \lambda \rrbracket \) iff

For all \( k, i \leq k < j \): Exists \( \lambda' \in Assig(\lambda) \): \( \langle \sigma[k], \sigma[k+1] \rangle \in R_{\lambda': [x \Rightarrow E]} \)

(4.2)

Thus, each pair of adjacent states in \( \sigma[i..j] \) models execution of some assignment statement and the corresponding changes to the target and control variables.

**Proof Outlines**

Having defined the program language, we now define the property language of Proof Outline Logic. A **proof outline** for a program \( \lambda \) associates an assertion with the opening and closing of each label in \( Lab(\lambda) \). The assertion associated with the opening of a label \( \lambda \) is called the **precondition** of \( \lambda \) and is denoted \( pre(\lambda) \); the assertion associated with its closing is called the **postcondition** of \( \lambda \) and is denoted \( post(\lambda) \).

Here is a grammar giving a syntax of proof outlines for our simple programming language.

\[
PO ::= \{p \} \lambda: [x \Rightarrow E] \{q\} \mid \{p\} \lambda: [PO_1; PO_2] \{q\} \mid \{p\} \lambda: [PO_1 // PO_2] \{q\}
\]

\( PO_1 \) and \( PO_2 \) are proof outlines, and \( p \) and \( q \) are assertions. A concrete example of a proof outline is given in Figure 4.3. It contains a proof outline for the program of Figure 4.1. Easier to read notations\(^{10}\) for proof outlines do exist; this format is particularly easy to define formally, so it is well

\[
\{true\}
\]

\[
\lambda: [\{true\} \\
\quad \lambda_1: [\{true\} \\
\quad \quad \lambda_{11}: [y := 1] \{y=1 \lor y=3\}; \\
\quad \quad \{y=1 \lor y=3\} \\
\quad \quad \lambda_{12}: [y := 2] \{y=2 \lor y=3\} \\
\quad \quad \} \{y=2 \lor y=3\} \\
\quad // \\
\quad \{true\} \\
\lambda_2: [y := 3] \{y=2 \lor y=3\} \\
\quad \} \{y=2 \lor y=3\}
\]

Figure 4.3. Example Proof Outline

---

\(^{10}\)For example, we sometimes write \( \{p\} PO(\lambda) \{q\} \) to denote a proof outline that is identical to \( PO(\lambda) \) but with \( p \) replacing \( pre(\lambda) \) and \( q \) replacing \( post(\lambda) \).
suited to our purpose.

Assertions in proof outlines are formulas of a first-order predicate logic. In this logic, terms and predicates are evaluated over traces, finite sequences of program states. A trace \( s_0 s_1 \ldots s_n \) that is a prefix of a program behavior defines a current program state \( s_n \) as well as a sequence \( s_0 s_1 \ldots s_{n-1} \) of past states. Thus, assertions interpreted with respect to traces can not only characterize the current state of the system, but can also characterize histories leading up to that state. Such expressiveness is necessary for proving arbitrary safety properties and for describing many environments.

The terms of our assertion language include constants, variables, the usual expressions over terms, and the past term \( \Theta T \) for \( T \) any term [S94].\(^\text{11}\) The \( \Theta \) operator allows terms to be constructed whose values depend on the past of a trace. For example, \( x + \Theta y \) evaluated in a trace \( s_0 s_1 s_2 \) equals \( s_2(x) + s_1(y) \). More formally, we define as follows the value \( \mathcal{M}[T]_\tau \) of a term \( T \) in trace \( \tau \), where \( c \) is a constant, \( v \) is a variable, and \( T_1 \) and \( T_2 \) are terms.

<table>
<thead>
<tr>
<th>term ( T )</th>
<th>( \mathcal{M}[T]_s s_1 \ldots s_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>( c )</td>
</tr>
<tr>
<td>( v )</td>
<td>( s_n(v) )</td>
</tr>
<tr>
<td>( T_1 + T_2 )</td>
<td>( \mathcal{M}[T_1]_s s_1 \ldots s_n + \mathcal{M}[T_2]_s s_1 \ldots s_n )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( \Theta T )</td>
<td>( \begin{cases} \mathcal{M}[T]<em>s s_1 \ldots s</em>{n-1} &amp; \text{if } n &gt; 0 \ \text{false} &amp; \text{if } n=0 \end{cases} )</td>
</tr>
</tbody>
</table>

Predicates of the assertion language are formed in the usual way from predicate symbols, terms, propositional connectives, and the universal and existential quantifiers. It is also convenient to regard Boolean-valued variables as predicates. This allows control variables to be treated as predicates. It also allows \( \Theta \text{true} \) to be treated as a predicate whose value is \( \text{true} \) in any trace having more than one state. Assertions are just predicates.

Proof outlines define properties. Informally, the property defined by a proof outline \( PO(\lambda) \) includes all behaviors \( \langle \sigma, i, j \rangle \) in which execution of \( \lambda \) starting in state \( \sigma[i] \) does not cause proof outline invariant \( I_{PO(\lambda)} \) to be invalidated. The proof outline invariant implies that the assertion associated with each control variable is \( \text{true} \) whenever that control variable is \( \text{true} \):

\[
I_{PO(\lambda)}: \lambda' \in Lab(\lambda) \land (\text{at}(\lambda') \Rightarrow \text{pre}(\lambda')) \land (\text{after}(\lambda') \Rightarrow \text{post}(\lambda')) \tag{4.3}
\]

It is easier to reason about proof outlines when the precondition for each statement \( \lambda' \) summarizes what is required for \( I_{PO(\lambda)} \) to hold when \( \text{at}(\lambda') \) is \( \text{true} \). For a proof outline \( PO(\lambda) \), this self consistency requirement is:

For every label \( \lambda' \in Lab(\lambda) \):

If \( \lambda' \) labels a sequential composition \( \lambda': [\lambda_1: [S_1]; \lambda_2: [S_2]] \) then:

\(^{11}\)The Proof Outline Logic of [S94] also allows recursively-defined terms using \( \Theta \). This increases the expressiveness of the assertion language, but is independent to the issues being addressed in this paper. Therefore, in the interest of simplicity, we omit such terms from the assertion language.
\[ pre(\lambda') \Rightarrow pre(\lambda_1) \]
\[ post(\lambda_1) \Rightarrow pre(\lambda_2) \]

If \( \lambda' \) labels a parallel composition \( \lambda' : [\lambda_1 : [S_1] // \lambda_2 : [S_2]] \) then:
\[ pre(\lambda') \Rightarrow (pre(\lambda_1) \land pre(\lambda_2)) \]

We can now formally define the set \( \llbracket PO(\lambda) \rrbracket \) of behaviors in the property \( PO(\lambda) \):
\[
\llbracket PO(\lambda) \rrbracket := \begin{cases} 
\emptyset & \text{if } PO(\lambda) \text{ is not self-consistent} \\
\{ (\sigma, i, j) \mid \sigma[..i] \equiv I_{PO(\lambda)} \text{ or for all } k, i \leq k \leq j: \sigma[..k] \equiv I_{PO(\lambda)} \} & \end{cases}
\]

(4.4)

Thus, \( \llbracket PO(\lambda) \rrbracket \) is empty if \( PO(\lambda) \) is not self-consistent. And, if \( PO(\lambda) \) is self-consistent, then \( \llbracket PO(\lambda) \rrbracket \) includes a behavior \( (\sigma, i, j) \) provided either (i) \( I_{PO(S)} \) is not satisfied when execution is started in state \( \sigma[i] \) or (ii) \( I_{PO(S)} \) is kept \( true \) throughout execution started in state \( \sigma[i] \). In the definition, proof outline invariant \( I_{PO(S)} \) is evaluated in prefixes of \( \sigma \) because assertions may contain terms involving \( \Theta \).

A proof outline is defined to be valid iff \( (\lambda, PO(\lambda)) \in Sat \) holds, where

\( (\lambda, PO(\lambda)) \in Sat \) if and only if \( \llbracket \lambda \rrbracket \subseteq \llbracket PO(\lambda) \rrbracket \)

as prescribed by (2.1). Appendix A contains a sound and complete proof system for establishing that a proof outline is valid. Such logics have become commonplace since Hoare’s original proposal [H69]. The particular axiomatization that we give is based on [S94], which, in turn, builds on the logic of [L80].

4.2. Exploiting an Environment with Proof Outlines

Our program language does not satisfy the closure conditions required for Program Reduction (2.3), nor does the property language (proof outlines) satisfy the closure conditions required for Property Reduction (2.4). To pursue property reduction, we define a language \( EnvL \) that characterizes properties imposed by environments. We then extend the property language so that it satisfies the necessary closure condition for property reduction.

We base \( EnvL \) on the assertion language of proof outlines. Every formula of \( EnvL \) is of the form \( \Box A \) where \( A \) is a formula of the assertion language. \( \Box A \) defines a set of behaviors as follows.

\[ \llbracket \Box A \rrbracket := \{ (\sigma, i, j) \mid \text{For all } k, i \leq k \leq j: \sigma[..k] \equiv A \} \]

Thus, \( \Box A \) contains behaviors \( (\sigma, i, j) \) for which prefixes \( \sigma[..i], \sigma[..i+1], \ldots, \sigma[..j] \) do not violate \( A \). Formulas in \( EnvL \) define safety properties, and \( EnvL \) includes all of the scheduler and real-time examples of §3.3 and §3.4. A more expressive assertion language (e.g. the one with recursive terms in [S94]) would enable all safety properties to be defined in this manner.

In order to close the property language of Proof Outline Logic under union with the complement of \( \llbracket \Box A \rrbracket \), we introduce a new form of proof outline. A constrained proof outline is a formula \( \Box A \rightarrow PO(\lambda) \), where \( A \) is a formula of the assertion language and \( PO(\lambda) \) is an ordinary proof outline. The property defined by a constrained proof outline is given by:

\[
\llbracket \Box A \rightarrow PO(\lambda) \rrbracket := \llbracket PO(\lambda) \rrbracket \cup \llbracket \Box A \rrbracket
\]

(4.6)

\[ \llbracket \Box A \rrbracket \] denotes the complement of \( \llbracket \Box A \rrbracket \). Generalizing from ordinary proof outlines, a constrained proof outline \( \Box A \rightarrow PO(\lambda) \) is considered valid iff \( (\lambda, \Box A \rightarrow PO(\lambda)) \in Sat \). Thus, if \( \Box A \rightarrow PO(\lambda) \) is
valid then $\llbracket\lambda\rrbracket \subseteq \llbracket\Box A \rightarrow PO(\lambda)\rrbracket$ holds.

The set of properties defined by constrained proof outlines and proof outlines does satisfy the necessary closure condition for property reduction. Given a program $\lambda$, let $L_\lambda$ be the set of constrained proof outlines and proof outlines for $\lambda$. The required closure condition is equivalent to:

**Lemma:** For any assertion $A$ and any $\Phi \in L_\lambda$, there exists a constrained proof outline $\Phi'$ in $L_\lambda$ such that

$\llbracket\Phi'\rrbracket = \llbracket\Phi\rrbracket \cup \llbracket\Box A\rrbracket$

**Proof.** The proof is by cases.

*Case: $\Phi$ is an ordinary proof outline.* In this case, choose $\Phi'$ to be $\Box A \rightarrow \Phi$.

*Case: $\Phi$ is a constrained proof outline $\Box B \rightarrow PO(\lambda)$.** In this case, choose $\Phi'$ to be $\Box (A \land B) \rightarrow PO(\lambda)$. This choice is justified by the following.

$\langle\sigma, i, j\rangle \in \llbracket\Box (A \land B) \rightarrow PO(\lambda)\rrbracket$

iff «definition (4.6) of $\llbracket\Box (A \land B) \rightarrow PO(\lambda)\rrbracket»

$\langle\sigma, i, j\rangle \in \langle\Box PO(\lambda)\rrbracket \cup \llbracket\Box (A \land B)\rrbracket\rangle$

iff «definition of $\llbracket\Box (A \land B)\rrbracket»

$\langle\sigma, i, j\rangle \in \langle\Box PO(\lambda)\rrbracket \cup \llbracket\Box A\rrbracket \cup \llbracket\Box B\rrbracket\rangle$

iff «definition (4.6) of $\llbracket\Box B \rightarrow PO(\lambda)\rrbracket»

$\langle\sigma, i, j\rangle \in \langle\Box B \rightarrow PO(\lambda)\rrbracket \cup \llbracket\Box A\rrbracket\rangle$

Q.E.D.

**Logic for Constrained Proof Outlines**

Our goal is to prove that a program $\lambda$ satisfies a property $PO(\lambda)$ under an environment $\Box A$:

$\langle\lambda, \Box A, PO(\lambda)\rangle \in ESat$ (4.7)

Using Property Reduction (2.4), we see that to prove (4.7), it suffices to be able to prove that $\lambda$ satisfies property $\Box A \rightarrow PO(\lambda)$.

$\langle\lambda, \Box A, PO(\lambda)\rangle \in ESat$

iff «Property Reduction (2.4)»

$\langle\lambda, PO(\lambda) \cup \Box A\rangle \in Sat$

iff «definition (2.1)»

$\llbracket\lambda\rrbracket \subseteq \llbracket PO(\lambda) \cup \Box A\rrbracket$

iff «$\llbracket F \cup G\rrbracket = \llbracket F\rrbracket \cup \llbracket G\rrbracket$ and definition (4.6) of $\Box A \rightarrow PO(\lambda)\rrbracket»

$\llbracket\lambda\rrbracket \subseteq \llbracket\Box A \rightarrow PO(\lambda)\rrbracket$

iff «definition (2.1)»

$\langle\lambda, \Box A \rightarrow PO(\lambda)\rangle \in Sat$

The deductive system of Appendix A enables us to prove that $\langle\lambda, \Phi\rangle \in Sat$ holds for $\Phi$ an ordinary proof outline. Extensions are needed for the case where $\Phi$ is a constrained proof outline. We now give these; a soundness and completeness proof for them appears in Appendix B.

For reasoning about assignment statements executed under an environment $\Box A$, we can assume that $A$ holds before execution and, because the environment precludes transition to a state satisfying $\neg A$, any postcondition asserting $\neg A$ can be strengthened.
Cnstr-Assig: \[
\{p \land A\} \quad \lambda: [x := E] \quad \{q \lor \neg A\}
\]
\[
\Box A \rightarrow \{p\} \quad \lambda: [x := E] \quad \{q\}
\]

Sequential composition under an environment \(\Box A\) allows a weaker postcondition for the first statement, since the environment ensures that \(A\) will hold.

Cnstr-SeqComp:
\[
\Box A \rightarrow PO(\lambda_1), \quad \Box A \rightarrow PO(\lambda_2)
\]
\[
(A \land post(\lambda_1)) \Rightarrow pre(\lambda_2)
\]
\[
\Box A \rightarrow \{pre(\lambda_1)\} \quad \lambda: \{PO(\lambda_1); PO(\lambda_2)\} \quad \{post(\lambda_2)\}
\]

Parallel composition under an environment \(\Box A\) also allows weaker assertions. \(A\) can be assumed in the preconditions of the interference-freedom proofs.

Cnstr-ParComp:
\[
\Box A \rightarrow PO(\lambda_1), \quad \Box A \rightarrow PO(\lambda_2),
\]
\[
\Box A \rightarrow PO(\lambda_1) \text{ and } \Box A \rightarrow PO(\lambda_2) \text{ are interference free}
\]
\[
\Box A \rightarrow \{pre(\lambda_1) \land pre(\lambda_2)\} \quad \lambda: \{PO(\lambda_1) \lor PO(\lambda_2)\} \quad \{post(\lambda_1) \land post(\lambda_2)\}
\]

We establish that \(\Box A \rightarrow PO(\lambda_1)\) and \(\Box A \rightarrow PO(\lambda_2)\) are interference free in much the same way as for ordinary proof outlines.

For all \(\lambda_\alpha \in Assig(\lambda_1)\), where \(\lambda_\alpha\) is the assignment \(\lambda_\alpha: [x := E]\):
\[
\Box A \rightarrow \{at(\lambda_\alpha) \land I_{PO(\lambda_1)} \land I_{PO(\lambda_2)}\} \quad \lambda_\alpha: [x := E] \quad \{I_{PO(\lambda_2)}\}
\]

For all \(\lambda_\alpha \in Assig(\lambda_2)\), where \(\lambda_\alpha\) is the assignment \(\lambda_\alpha: [x := E]\):
\[
\Box A \rightarrow \{at(\lambda_\alpha) \land I_{PO(\lambda_2)} \land I_{PO(\lambda_1)}\} \quad \lambda_\alpha: [x := E] \quad \{I_{PO(\lambda_1)}\}
\]

As with ordinary proof outlines, two rules allow us to modify assertions based on deductions possible in the assertion language. For a constrained proof outline \(\Box A \rightarrow PO(\lambda)\), we can assume \(A\) in making those deductions.

Cnstr-Conseq:
\[
\Box A \rightarrow PO(\lambda), \quad (p \land A) \Rightarrow pre(\lambda), \quad (post(\lambda) \land A) \Rightarrow \{q\}
\]
\[
\Box A \rightarrow \{p\} \quad PO(\lambda) \quad \{q\}
\]

Cnstr-Equiv:
\[
\Box A \rightarrow PO(\lambda), \quad A \Rightarrow (I_{PO(\lambda)} \land I_{PO(\lambda)}), \quad PO'(\lambda) \text{ is self consistent}
\]
\[
\Box A \rightarrow PO'(\lambda)
\]

Example Revisited

We illustrate the deductive system for constrained proof outlines by proving that \(y=3\) holds upon termination, when the program of Figure 4.1 is executed by a single processor using a fixed-priority scheduler with process \(\lambda_1\) having higher priority than \(\lambda_2\).

Recall that a fixed-priority scheduler rules out allocating a processor to any but the highest-priority processes, where a fixed priority value \(v_\pi\) is associated with each process \(\pi\). The formulation of this restriction using the assertion language of our Proof Outline Logic closely parallels our TLA
formulation in §3.3.

As before, for $N$ the number of processors, we define:

$\text{Alloc}(N)$: $(\#\pi \in \Pi: \text{active}_\pi) \leq N$

$\text{Run}(\pi)$: $\text{active}_\pi \Rightarrow \pi \in \text{TOP}(N, \Pi)$

These state that variable $\text{active}_\pi$ is true for the $N$ highest ranked different processes $\pi$. To stipulate that $\text{active}_\pi$ be true in order for a process to execute an atomic action, let $\text{Lab}(\lambda_\pi)$ be the set of labels for process $\pi$. Execution of an atomic action from $\pi$ causes control variables to change for some $\lambda' \in \text{Lab}(\lambda_\pi)$.

$\text{Pgrs}(\pi)$: $(\Theta \text{true} \land \forall \lambda' \in \text{Lab}(\lambda_\pi) (at(\lambda') \neq \Theta at(\lambda'))) \Rightarrow \Theta \text{active}_\pi$

The rank $\text{rank}_\pi$ of a process depends on whether or not that process has terminated. Since we assume that process $\pi$ has label $\lambda_\pi$, that process has not terminated if $\text{in}(\lambda_\pi)$ is true. We thus can assign values to $\text{rank}_\pi$ using $\nu_\pi$ as follows.

$\text{Prio}(\pi)$: $(\text{in}(\lambda_\pi) \Rightarrow (\text{rank}_\pi = \nu_\pi)) \land (\neg \text{in}(\lambda_\pi) \Rightarrow (\text{rank}_\pi = 0))$

Combining these, we obtain an assertion $\text{FixedPrio}$ which characterizes a fixed-priority scheduler.

$\text{FixedPrio}$: $\text{Alloc}(N) \land (\forall \pi \in \Pi: \text{Run}(\pi) \land \text{Pgrs}(\pi) \land \text{Prio}(\pi))$

To conclude that $y = 3$ holds upon termination of program $\lambda$ in Figure 4.1, we prove $\Box \text{FixedPrio} \rightarrow \text{PO}(\lambda)$ a theorem, where $\text{post}(\lambda) \Rightarrow y = 3$. We assume $N = 1$, $\nu_{\lambda_1} = 2$, and $\nu_{\lambda_2} = 1$.

Using Assig2 (of Appendix A), we get:

\begin{align*}
\{ \text{at}(\lambda_2) \} \lambda_{11} & : [y := 1] \{ \text{at}(\lambda_2) \} \\
\{ \text{at}(\lambda_2) \} \lambda_{12} & : [y := 2] \{ \text{at}(\lambda_2) \}
\end{align*}

With Conseq (of Appendix A), we can strengthen the precondition of (4.8) and (4.9) as well as weakening the postconditions of both—in preparation for using Cnstr-Assig with $\Box \text{FixedPrio}$

\begin{align*}
\{ \text{at}(\lambda_2) \land \text{FixedPrio} \} \lambda_{11} & : [y := 1] \{ \text{at}(\lambda_2) \lor \neg \text{FixedPrio} \} \\
\{ \text{at}(\lambda_2) \land \text{FixedPrio} \} \lambda_{12} & : [y := 2] \{ \text{true} \lor \neg \text{FixedPrio} \}
\end{align*}

Using Cnstr-Assig we now obtain:

$\Box \text{FixedPrio} \rightarrow \{ \text{at}(\lambda_2) \} \lambda_{11} : [y := 1] \{ \text{at}(\lambda_2) \}$

$\Box \text{FixedPrio} \rightarrow \{ \text{at}(\lambda_2) \} \lambda_{12} : [y := 2] \{ \text{true} \}$

We combine these, using Cnstr-SeqComp to obtain a constrained proof outline for process $\lambda_1$.

$\Box \text{FixedPrio} \rightarrow \{ \text{at}(\lambda_2) \}$

\begin{align*}
\lambda_1 & : [\{ \text{at}(\lambda_2) \}] \\
\lambda_{11} & : [y := 1] \{ \text{at}(\lambda_2) \} \\
\lambda_{12} & : [y := 2] \{ \text{true} \}
\end{align*}

A proof outline for process $\lambda_2$ is constructed by starting with Assig1 (of Appendix A).

$\{3 = 3\} \lambda_2 : [y := 3] \{ y = 3 \}$

In preparation for using Cnstr-Assig, the precondition is strengthened and postcondition is weakened.
\{true \land FixedPrio\} \quad \lambda_2: [y := 3] \quad \{y = 3 \lor \neg FixedPrio\}

We now can use Cnstr-Assig to obtain a constrained proof outline for process \( \lambda_2 \).

\[ \square FixedPrio \rightarrow \{true\} \quad \lambda_2: [y := 3] \quad \{y = 3\} \]

(4.17)

Finally, we use Cnstr-ParComp to combine (4.14) and (4.17):

\[ \square FixedPrio \rightarrow \{at(\lambda_2)\} \]
\[ \lambda: [\{at(\lambda_2)\}] \]
\[ \lambda_1: [\{at(\lambda_2)\} \quad \lambda_{11}: [y := 1] \quad \{at(\lambda_2)\} \quad \lambda_{12}: [y := 2] \quad \{true\}] \]
\[ \{true\} \quad \lambda_2: [y := 3] \quad \{y = 3\}] \]

(4.18)

\[ [y = 3] \]

This requires that we discharge the following interference-freedom requirements:

\[ \square FixedPrio \rightarrow \{at(\lambda_{11}) \land I_{PO(\lambda_1)} \land I_{PO(\lambda_2)}\} \quad \lambda_{11}: [y := 1] \quad \{I_{PO(\lambda_2)}\} \]

(4.19)

\[ \square FixedPrio \rightarrow \{at(\lambda_{12}) \land I_{PO(\lambda_1)} \land I_{PO(\lambda_2)}\} \quad \lambda_{12}: [y := 2] \quad \{I_{PO(\lambda_2)}\} \]

(4.20)

\[ \square FixedPrio \rightarrow \{at(\lambda_2) \land I_{PO(\lambda_2)} \land I_{PO(\lambda_1)}\} \quad \lambda_2: [y := 3] \quad \{I_{PO(\lambda_1)}\} \]

(4.21)

where:

\[ I_{PO(\lambda_1)}: \quad (at(\lambda_1) \Rightarrow at(\lambda_2)) \land (after(\lambda_1) \Rightarrow true) \]
\[ \land (at(\lambda_{11}) \Rightarrow at(\lambda_2)) \land (after(\lambda_{11}) \Rightarrow at(\lambda_2)) \]
\[ \land (at(\lambda_{12}) \Rightarrow at(\lambda_2)) \land (after(\lambda_{12}) \Rightarrow true) \]

\[ I_{PO(\lambda_2)}: \quad (at(\lambda_2) \Rightarrow true) \land (after(\lambda_2) \Rightarrow y = 3) \]

\( I_{PO(\lambda_1)} \) and \( I_{PO(\lambda_2)} \) can be simplified, using ordinary Predicate Logic, resulting in:

\[ I_{PO(\lambda_1)}: \quad (at(\lambda_1) \lor at(\lambda_{11}) \lor after(\lambda_{11}) \lor at(\lambda_{12})) \Rightarrow at(\lambda_2) \]

\[ I_{PO(\lambda_2)}: \quad after(\lambda_2) \Rightarrow y = 3 \]

To prove formula (4.19), observe that according to the definitions of \( I_{PO(\lambda_1)} \), \( I_{PO(\lambda_2)} \), and \( FixedPrio \):

\[ (at(\lambda_{11}) \land I_{PO(\lambda_1)} \land I_{PO(\lambda_2)} \land FixedPrio) \Rightarrow at(\lambda_2) \]

\[ at(\lambda_2) \Rightarrow (I_{PO(\lambda_2)} \lor \neg FixedPrio) \]

Applying Conseq and then Cnstr-Assig to (4.8) we obtain (4.19). The proof of (4.20) is virtually identical.

Proving formula (4.21) illustrates the role of environment \( \square FixedPrio \). Using Assig3, Equiv, and Conseq it is not difficult to prove:

\[ \{at(\lambda_2)\} \quad \lambda_2: [y := 3] \quad \{Q: \ \Theta at(\lambda_2) \land \neg at(\lambda_2) \land at(\lambda_1) \land (in(\lambda_1) \lor after(\lambda_1))\} \]

\[ \{in(\lambda_1) \Rightarrow (active_{\lambda_1} \land \neg active_{\lambda_2})\} \quad \lambda_2: [y := 3] \quad \{R: \ \Theta (in(\lambda_1) \Rightarrow (active_{\lambda_1} \land \neg active_{\lambda_2}))\} \]

Each of these preconditions is implied by \( at(\lambda_2) \land I_{PO(\lambda_2)} \land I_{PO(\lambda_1)} \land FixedPrio \), so we can use
Conseq to strengthen each and deduce:

\[
\{ at(\lambda_2) \land I_{P_{O}(\alpha_2)} \land I_{P_{O}(\alpha_3)} \land FixedPrio \} \quad \lambda_2: [y := 3] \quad \{ Q \}
\]

\[
\{ at(\lambda_2) \land I_{P_{O}(\alpha_2)} \land I_{P_{O}(\alpha_3)} \land FixedPrio \} \quad \lambda_2: [y := 3] \quad \{ R \}
\]

Therefore, by Conj, we obtain:

\[
\{ at(\lambda_2) \land I_{P_{O}(\alpha_2)} \land I_{P_{O}(\alpha_3)} \land FixedPrio \} \quad \lambda_2: [y := 3] \quad \{ Q \land R \}
\]

We now use Conseq to infer that \( I_{P_{O}(\alpha_3)} \) or \( \neg FixedPrio \) holds whenever \( Q \land R \) does by proving:

\[
\text{(in}(\lambda_1) \land Q \land R) \Rightarrow \neg FixedPrio
\]

\[
\text{(after}(\lambda_1) \land Q \land R) \Rightarrow I_{P_{O}(\alpha_3)}
\]

Using these with Conseq, we conclude:

\[
\{ at(\lambda_2) \land I_{P_{O}(\alpha_2)} \land I_{P_{O}(\alpha_3)} \land FixedPrio \} \quad \lambda_2: [y := 3] \quad \{ I_{P_{O}(\alpha_3)} \lor \neg FixedPrio \}
\]

Cnstr-Assig now allows us to conclude (4.21), as is desired.

### 4.3. An Even Older Recipe

The notion of a constrained proof outline is not new. In [LS85] a similar idea was discussed in connection with reasoning about aliasing and other artifacts of variable declarations. The aliasing of two variables imposes the constraint that their values are equal; the declaration of a variable imposes a constraint on the values that variable may store. Constrained proof outlines, because they provide a basis for proving properties of programs whose execution depends on constraints being preserved, are thus a way to reason about aliasing and declarations. An even earlier call for a construct like our constrained proof outlines appears in [L80]. There, Lamport claims that such proof outlines would be helpful in proving certain types of safety properties of concurrent programs.

### 5. Discussion

**Related Work**

Our work is perhaps closest in spirit to the various approaches for reasoning about open systems. An open system is one that interacts with its environment through shared memory or communication. The execution of such a system is commonly modeled as an interleaving of steps by the system and steps by the environment. Since an open system is not expected to function properly in an arbitrary environment, its specification typically will contain explicit assumptions about the environment. Such specifications are called assume-guarantee specifications because they guarantee behavior when the environment satisfies some assumptions. Logics for verifying safety properties of assume-guarantee specifications are discussed in [FFG92], [J83], and [MC81]; liveness properties are treated in [AL91], [BKP84], and [P85]; and model-checking techniques based on assume-guarantee specifications are introduced in [CLM89] and [GL91].

Our approach differs from this open systems work both in the role played by the environment and in how state changes are made by the environment. We use the environment to represent aspects of the computation model, not as an abstraction of the behaviors for other agents that will run concurrently with the system. Second, in our approach, every state change obeys constraints defined by the environment. State changes attributed to the environment are not interleaved with system actions, as is the case with the open systems view.
Our view of the environment and the view employed for open systems are complementary. They address different problems. Both notions of environment can coexist in a single logic. Open systems and their notion of an environment are an accepted part of the verification scene. This paper explores the use of a new type of environment. Our environments allow logics to be extended for various computational models. As a result, a single principle suffices for reasoning about the effects of schedulers, real-time models, resource constraints, and fairness assumptions. Thus, one does not have to redesign a programming logic every time the computational model is changed.

In terms of program construction, our notion of an environment is closely related to superposition [K87] [BF88] [CM88]. The superposition of two programs S and T is a single program, each of whose steps comprises a step of S and a step of T performed simultaneously. Thus, in terms of TLA, the superposition of two actions is simply their conjunction. Our work extends the domain of applicability for superposition by allowing one component of a superposition to characterize aspects of a computational model.

Redefining Feasible Behaviors

The definition of §2 for the feasible behaviors of a program S under an environment E is not the only plausible one. Every infeasible behavior of S ruled out by E has a maximal finite prefix (possibly empty) that agrees with a prefix of some behavior in E. Such a prefix can be regarded as modeling an execution of S that aborts due to the constraints of E, and this prefix might well be included in the set of feasible behaviors.

For example, consider executing the program

\[ T: \ x := 0; \quad \textbf{do} \ i := 1 \textbf{ to } 5: \ x := x + 1 \quad \textbf{od} \]

in an environment that constrains x to be between 0 and 3 (i.e., x is represented using 2 bits). The alternative definition of feasible behaviors would include prefixes of behaviors of T up until the point where an attempt is made to store 4 into x. Using the definition of §2, the set of feasible behaviors would be empty.

The alternative definition of feasible behaviors for a program S under an environment E,

\[
(\llbracket S \rrbracket \land \llbracket E \rrbracket) \cup (\text{prefix}(\llbracket S \rrbracket) \land \text{prefix}(\llbracket E \rrbracket)),
\]  

admits reasoning about feasible, but incomplete, executions of a program under a given environment. Unfortunately, we have been unable to identify reduction principles for definition (5.1). It remains an open question how to extend a logic for Sat into a logic for ESat given this definition.

6. Conclusion

In this paper, we have shown that environments are a powerful device for making aspects of a computational model explicit in a programming logic. We have shown how environments can be used to formalize schedulers and real-time; a forthcoming paper will show how they can be applied to hybrid systems, where a continuous transition system governs changes to certain variables.

We have given two semantic principles, program reduction and property reduction, for extending programming logics to enable reasoning about program executions feasible under a specified environment. Having such principles means that a new logic need not be designed every time the computational model served by an extant logic is changed. For example, in this paper, we give a new way to reason about real-time in TLA and in Hoare-style programming logics. We also derive the
first Hoare-style logic for reasoning about schedulers.

The basic idea of reasoning about program executions that are feasible in some environment is not new, having enjoyed widespread use in connection with open systems. The basic idea of augmenting the individual state transitions caused by the atomic actions in a program is not new, either. It underlies methods for program composition by superposition, methods for reasoning about aliasing, and proposals for verifying certain types of safety properties. What is new is our use of environments for describing aspects of a computational model and our unifying semantic principles for reasoning about environments. Extensions to a computational model can now be translated into extensions to an existing programming logic, by applying one of two simple semantic principles.

Acknowledgments
We are grateful to Leslie Lamport and Jay Misra for discussions and helpful comments on this material.

References
[LS84] Lamport, L. and F.B. Schneider. The "Hoare Logic" of CSP and all that. ACM TOPLAS 6, 2 (April 1984), 281-296.


Appendix A: A Logic of Proof Outlines

The deductive system for reasoning about assertions includes the axioms and inference rules of first-order predicate logic. It also axiomatizes theories for the datatypes of program variables and expressions. Perhaps the only aspect of this axiomatization that might be unfamiliar concerns $\Theta$. It will include axioms like:

$\Theta true \Rightarrow (\Theta\mathcal{E}(I_1, \ldots, I_n) = \mathcal{E}(\Theta I_1, \ldots, \Theta I_n))$

In order to reason about control variables in program states, each program $\lambda$ gives rise to a set of axioms. These axioms characterize the constraints of Figure 4.2. For every label $\lambda' \in Lab(\lambda)$:

CP0: \(\text{in}(\lambda) \neq \text{after}(\lambda)\)

CP1: \(\neg(\text{at}(\lambda') \land \text{after}(\lambda'))\)

CP2: \(\text{at}(\lambda') \Rightarrow \text{in}(\lambda')\)

CP3: If $\lambda'$ labels $[x := E]$: \(\text{at}(\lambda') = \text{in}(\lambda')\)

CP4: If $\lambda'$ labels $[\lambda_1; S; \lambda_2; T]]$:
   (a) \(\text{at}(\lambda') = \text{at}(\lambda_1)\)
   (b) \(\text{at}(\lambda_2) = \text{after}(\lambda_1)\)
   (c) \(\text{after}(\lambda') = \text{after}(\lambda_2)\)
   (d) \(\text{in}(\lambda') = ((\text{in}(\lambda_1) \land \neg \text{in}(\lambda_2)) \lor (\neg \text{in}(\lambda_1) \land \text{in}(\lambda_2)))\)
   (e) \(\text{in}(\lambda_1) \lor \text{in}(\lambda_2) \Rightarrow \text{in}(\lambda')\)

CP5: If $\lambda'$ labels $[\lambda_1; S//\lambda_2; T]]$:
   (a) \(\text{at}(\lambda') = (\text{at}(\lambda_1) \land \text{at}(\lambda_2))\)
   (b) \(\text{after}(\lambda') = (\text{after}(\lambda_1) \land \text{after}(\lambda_2))\)
   (d) \(\text{in}(\lambda') = ((\text{in}(\lambda_1) \lor \text{after}(\lambda_1)) \land (\text{in}(\lambda_2) \lor \text{after}(\lambda_2)) \land \neg (\text{after}(\lambda_1) \land \text{after}(\lambda_2)))\)
   (d) \(\text{in}(\lambda_1) \lor \text{in}(\lambda_2) \Rightarrow \text{in}(\lambda')\)

For reasoning about proof outlines, we have the following. First, here are the axioms for assignment statements.

Assig1: If no free variable in $p$ is a control variable and $p^E_{\lambda}$ denotes the predicate logic formula that results from replacing every free occurrence of $x$ in $p$ that is not in the scope of $\Theta$ with $E$:

\[
\{p^E_{\lambda}\} \lambda: [x := E] \{p\}
\]

Assig2: If $\lambda'$ is a label from a program that is parallel to that containing $\lambda$, and $cp(\lambda')$ denotes any of the control variables $\text{at}(\lambda'), \text{after}(\lambda'), \text{in}(\lambda')$ or their negations:

\[
\{cp(\lambda')\} \lambda: [x := E] \{cp(\lambda')\}
\]

Assig3: \(\{p\} \lambda: [x := E] \{\Theta p\}\)

Sequential composition is handled by a single inference rule.
SeqComp: \[ PO(\lambda_1), \ PO(\lambda_2), \]
\[ \text{pre}(\lambda_1) \Rightarrow \text{pre}(\lambda_2) \]
\[ \{ \text{pre}(\lambda_1), \ \lambda: [ PO(\lambda_1); \ PO(\lambda_2) ] \ \{ \text{post}(\lambda_2) \} \]

The parallel composition rule is based on the formulation of interference freedom [OG76] of proof outlines given in [LS84]. Two proof outlines \( PO(\lambda_1) \) and \( PO(\lambda_2) \) are interference free iff

For all \( \lambda_\alpha \in \text{Assign}(\lambda_1) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [ x := E ]: \)
\[ \{ \text{at}(\lambda_\alpha) \land I_{PO(\lambda_\alpha)} \land I_{PO(\lambda_2)} \} \ \lambda_\alpha: [ x := E ] \ \{ I_{PO(\lambda_2)} \} \]

For all \( \lambda_\alpha \in \text{Assign}(\lambda_2) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [ x := E ]: \)
\[ \{ \text{at}(\lambda_\alpha) \land I_{PO(\lambda_2)} \land I_{PO(\lambda_1)} \} \ \lambda_\alpha: [ x := E ] \ \{ I_{PO(\lambda_1)} \} \]

ParComp: \[ PO(\lambda_1), \ PO(\lambda_2), \]
\[ \text{PO}(\lambda) \text{ and PO}(\lambda) \text{ are interference free} \]
\[ \{ \text{pre}(\lambda_1) \land \text{pre}(\lambda_2) \} \ \lambda: [ \text{PO}(\lambda_1) \lor \text{PO}(\lambda_2) ] \ \{ \text{post}(\lambda_1) \land \text{post}(\lambda_2) \} \]

Finally, three rules allow us to modify assertions based on the deductive system for the assertion language. Recall, \( \{ p \} \ PO(\lambda) \ { q \} \) denotes a proof outline that is identical to \( PO(\lambda) \) but with \( p \) replacing \( pre(\lambda) \) and \( q \) replacing \( post(\lambda) \).

Conj: \[ \{ p_1 \} \ \lambda: [ x := E ] \ \{ q_1 \} \]
\[ \{ p_2 \} \ \lambda: [ x := E ] \ \{ q_2 \} \]
\[ \{ p_1 \land p_2 \} \ \lambda: [ x := E ] \ \{ q_1 \land q_2 \} \]

Conseq: \[ PO(\lambda), \ p \Rightarrow \text{pre}(\lambda), \ \text{post}(\lambda) \Rightarrow q \]
\[ \{ p \} \ PO(\lambda) \ { q \} \]

Equiv: \[ PO(\lambda), \ I_{PO(\lambda)} = I_{PO'(\lambda)}, \ I_{PO'(\lambda)} \text{ self consistent} \]
\[ \text{PO}'(\lambda) \]

Appendix B: Soundness and Completeness for Constrained Proof Outlines

We now prove soundness and relative completeness for the Logic of Constrained Proof Outlines given in section 4.2. Specifically, we prove that Cnstr-Assig, Cnstr-SeqComp, Const-ParComp and Cnstr-Equiv are sound. We also prove that Cnstr-Assig, Cnstr-SeqComp and Const-ParComp comprise a complete deductive system relative to the deductive system of Appendix A for ordinary proof outlines. (Cnstr-Conseq and Cnstr-Equiv of section 4.2 are not necessary for completeness.)
Lemma (Soundness of Cnstr-Assig): The rule

\[
\begin{align*}
\text{Cnstr-Assig:} & \quad \frac{p \land A \quad \lambda: [x := E] \; q \lor \neg A}{\Box A \to \{p\} \; \lambda: [x := E] \; \{q\}}
\end{align*}
\]

is sound.

Proof. Assume that hypothesis \(\{p \land A\} \lambda: [x := E] \; \{q \lor \neg A\}\) is valid. We show that if \(\langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket\) holds, then \(\langle \sigma, i, j \rangle \in \llbracket \Box A \to \{p\} \lambda: [x := E] \; \{q\} \rrbracket\) holds, and thus that \(\Box A \to \{p\} \lambda: [x := E] \; \{q\}\) is valid.

If \(\{p \land A\} \lambda: [x := E] \; \{q \lor \neg A\}\) is valid then, by definition, \(\llbracket \lambda \rrbracket \subseteq \llbracket PO(\lambda) \rrbracket\) holds, where \(PO(\lambda)\) is \(\{p \land A\} \lambda: [x := E] \; \{q \lor \neg A\}\). This implies that for any \(\langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket\) one of the following must hold:

\[
\sigma[i\neq k]\#I_{PO(\lambda)} \tag{B.1.1}
\]

For all \(k\), \(i \leq k \leq j\) : \(\sigma[k]=I_{PO(\lambda)} \tag{B.1.2}\)

where \(I_{PO(\lambda)}\): \((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A)\)

We consider two cases.

Case 1: Assume \(\langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket\). According to definition (4.6), \(\langle \sigma, i, j \rangle\) is in \(\llbracket \Box A \to \{p\} \lambda: [x := E] \; \{q\} \rrbracket\).

Case 2: Assume \(\langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket\). According to the definition of \(\llbracket \Box A \rrbracket\):

For all \(k\), \(i \leq k \leq j\) : \(\sigma[k]=A \tag{B.1.3}\)

It suffices to prove that if (B.1.1) holds or (B.1.2) holds, then \(\langle \sigma, i, j \rangle \in \llbracket \Box A \to \{p\} \lambda: [x := E] \; \{q\} \rrbracket\).

Case 2.1: Assume (B.1.1) holds. Thus

\(\sigma[i\neq k]=((at(\lambda) \land \neg \neg p \lor \neg A) \lor (after(\lambda) \land \neg q \land A))\)

holds. Conjoining (B.1.3), we conclude

\(\sigma[k]=((at(\lambda) \land \neg p) \lor (after(\lambda) \land \neg q))\)

which implies \(\sigma[k]\neq at(\lambda) \Rightarrow p\) \land (after(\lambda) \Rightarrow q)\). Because \(\{p\} \lambda: [x := E] \; \{q\}\) is self consistent, by definition (4.4) we have that \(\langle \sigma, i, j \rangle \in \llbracket \{p\} \lambda: [x := E] \; \{q\} \rrbracket\) holds. Hence, by definition (4.6) of the property defined by a constrained proof outline, \(\langle \sigma, i, j \rangle \in \llbracket \Box A \to \{p\} \lambda: [x := E] \; \{q\} \rrbracket\) holds.

Case 2.2: Assume (B.1.2) holds. Conjoining (B.1.3), we conclude

For all \(k\), \(i \leq k \leq j\) : \(\sigma[k]=((at(\lambda) \Rightarrow p) \land (after(\lambda) \Rightarrow q))\).

Because \(\{p\} \lambda: [x := E] \; \{q\}\) is self consistent, by definition (4.4) we have that \(\langle \sigma, i, j \rangle \in \llbracket \{p\} \lambda: [x := E] \; \{q\} \rrbracket\) holds, so by definition (4.6) , \(\langle \sigma, i, j \rangle \in \llbracket \Box A \to \{p\} \lambda: [x := E] \; \{q\} \rrbracket\) holds as well. \(\square\)
Lemma (Soundness of Cnstr-SeqComp): The rule

\[
\begin{align*}
\Box A \rightarrow PO(\lambda_1), \quad \Box A \rightarrow PO(\lambda_2) \\
(A \land \text{post}(\lambda_1)) \Rightarrow \text{pre}(\lambda_2)
\end{align*}
\]

\[
\Box A \rightarrow \{\text{pre}(\lambda_1)\} \quad \lambda: [PO(\lambda_1); PO(\lambda_2)] \{\text{post}(\lambda_2)\}
\]

is sound.

Proof. Assume that the hypotheses are valid. Therefore, we have that \(PO(\lambda_1)\) is self consistent, \(PO(\lambda_2)\) is self consistent, and:

\[
\begin{align*}
\llbracket \lambda_1 \rrbracket \subseteq (\llbracket PO(\lambda_1) \rrbracket \cup \llbracket \Box A \rrbracket) \\
\llbracket \lambda_2 \rrbracket \subseteq (\llbracket PO(\lambda_2) \rrbracket \cup \llbracket \Box A \rrbracket) \tag{B.2.1}
\end{align*}
\]

\[
\begin{align*}
(A \land \text{post}(\lambda_1)) \Rightarrow \text{pre}(\lambda_2) \tag{B.2.2}
\end{align*}
\]

To establish validity of the rule’s conclusion, we must prove that \(\llbracket \lambda \rrbracket \subseteq \llbracket \Box A \rightarrow PO(\lambda) \rrbracket\), where \(PO(\lambda)\) is \(\{\text{pre}(\lambda_1)\} \quad \lambda: [PO(\lambda_1); PO(\lambda_2)] \{\text{post}(\lambda_2)\}\). We do this by proving \((\sigma, i, j)\in\llbracket \lambda \rrbracket\) implies \((\sigma, i, j)\in\llbracket PO(\lambda) \rrbracket \cup \llbracket \Box A \rrbracket\), where, according to definition (4.3) of \(I_{PO(\lambda)}\), we have:

\[
I_{PO(\lambda)}: \quad I_{PO(\lambda_1)} \land I_{PO(\lambda_2)} \land (\text{at}(\lambda) \Rightarrow \text{pre}(\lambda_1)) \land (\text{after}(\lambda) \Rightarrow \text{post}(\lambda_2))
\]

Let \((\sigma, i, j)\in\llbracket \lambda \rrbracket\) hold. We consider two cases.

Case 1: Assume \((\sigma, i, j)\in\llbracket \Box A \rrbracket\). According to definition (4.6) of a constrained proof outline, \((\sigma, i, j)\in\llbracket \Box A \rightarrow PO(\lambda) \rrbracket\) holds.

Case 2: Assume \((\sigma, i, j)\in\llbracket \Box A \rrbracket\). In order to conclude \((\sigma, i, j)\in\llbracket \Box A \rightarrow PO(\lambda) \rrbracket\) holds, we must show that \((\sigma, i, j)\in\llbracket PO(\lambda) \rrbracket\) holds. According to the definition of sequential composition \(\llbracket \lambda: [\lambda_1; \lambda_2] \rrbracket\), there are three cases:

\[
\begin{align*}
(2.1) \quad \text{For all } k, i \leq k \leq j: \quad \sigma[k] \#(\text{in}(\lambda_2) \lor \text{after}(\lambda_2)) \quad \text{and} \quad (\sigma, i, j) \in \llbracket \lambda_1 \rrbracket. \\
(2.2) \quad \text{For all } k, i \leq k \leq j: \quad \sigma[k] \#(\text{in}(\lambda_1) \lor \text{after}(\lambda_1)) \quad \text{and} \quad (\sigma, i, j) \in \llbracket \lambda_2 \rrbracket. \\
(2.3) \quad \text{Exists } n, i \leq n \leq j: \quad (\sigma, i, n) \in \llbracket \lambda_1 \rrbracket \quad \text{and} \quad (\sigma, n, j) \in \llbracket \lambda_2 \rrbracket \quad \text{and} \\
\sigma[n] = (\text{after}(\lambda_1) \land \text{at}(\lambda_2)) \quad \text{and} \\
\text{for all } k, 1 \leq k < n: \quad \sigma[k] \#(\text{in}(\lambda_2) \lor \text{after}(\lambda_2)) \quad \text{and} \\
\text{for all } k, n < k \leq i: \quad \sigma[k] \#(\text{in}(\lambda_1) \lor \text{after}(\lambda_1))
\end{align*}
\]

Case 2.1: Assume that for all \(k, i \leq k \leq j: \sigma[k] \#(\text{in}(\lambda_2) \lor \text{after}(\lambda_2))\) and \((\sigma, i, j)\in\llbracket \lambda_1 \rrbracket\) hold. According to (B.2.1), \((\sigma, i, j)\in(\llbracket PO(\lambda_1) \rrbracket \cup \llbracket \Box A \rrbracket)\). Given assumption (of Case 2) \((\sigma, i, j)\in\llbracket \Box A \rrbracket\), we conclude \((\sigma, i, j)\in\llbracket PO(\lambda_1) \rrbracket\). Thus, by definition (4.4) of \(\llbracket PO(\lambda_1) \rrbracket\), we have that either \(\sigma[\ldots i] \in I_{PO(\lambda_i)}\) or else for all \(k, i \leq k \leq j: \sigma[\ldots k] = I_{PO(\lambda_i)}\). Since \(\text{at}(\lambda_1) = \text{at}(\lambda)\) due to Control Predicate Axiom CP4(a) of Appendix A and \(\text{pre}(\lambda)\) is \(\text{pre}(\lambda_1)\), we conclude that either

\[
\sigma[\ldots i] \#(I_{PO(\lambda_i)} \land (\text{at}(\lambda) \Rightarrow \text{pre}(\lambda_1))) \quad \text{or else} \tag{B.2.5}
\]

-27-
for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda_i)} \land (at(\lambda) \Rightarrow pre(\lambda_1))$. \hfill (B.2.6)

From assumption (of Case 2.1) for all $k, i \leq k \leq j$: $\sigma[k] \not\models (in(\lambda_2) \lor after(\lambda_2))$, we conclude that for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda_i)}$. And, because $after(\lambda_2) = after(\lambda)$ due to Control Predicate Axiom CP4(c) of Appendix A, we have for all $k, i \leq k \leq j$: $\sigma[..k] = (I_{PO(\lambda)} \land (after(\lambda) \Rightarrow post(\lambda_2)))$. Conjoining this with (B.2.5) and (B.2.6) we get that either $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda)}$ holds. Since $PO(\lambda)$ is self consistent (because $(A \land post(\lambda_1)) \Rightarrow pre(\lambda_2)$ and both $PO(\lambda_1)$ and $PO(\lambda_2)$ are self consistent), we have $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda) \rrbracket$. Thus, by definition (4.6) we conclude that $\langle \sigma, i, j \rangle \in \llbracket \square A \rightarrow PO(\lambda) \rrbracket$.

Case 2.2: Assume for all $k, i \leq k \leq j$: $\sigma[k] \not\models (in(\lambda_1) \lor after(\lambda_1))$ and $\langle \sigma, i, j \rangle \in \llbracket \lambda_2 \rrbracket$ hold. According to (B.2.2), $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda_2) \rrbracket \cup \llbracket \square A \rrbracket$. Given assumption (of Case 2) $\langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket$, we conclude $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda_2) \rrbracket$. Thus, by definition (4.4) of $\llbracket PO(\lambda_2) \rrbracket$, we have that either $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda)}$. Since $after(\lambda_2) = after(\lambda)$ due to Control Predicate Axiom CP4(c) of Appendix A, we conclude that either

$$\sigma[..i] \not\models (I_{PO(\lambda)} \land (after(\lambda) \Rightarrow post(\lambda_2))) \quad \text{or else}$$

for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda)} \land (after(\lambda) \Rightarrow post(\lambda_2))$. \hfill (B.2.7)

(B.2.8)

From assumption (of Case 2.2) for all $k, i \leq k \leq j$: $\sigma[k] \not\models (in(\lambda_1) \lor after(\lambda_1))$ we conclude that for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda_i)}$. And, because $at(\lambda_1) = at(\lambda)$ due to Control Predicate Axiom CP4(a) of Appendix A, we have for all $k, i \leq k \leq j$: $\sigma[..k] = (I_{PO(\lambda_i)} \land (at(\lambda_1) \Rightarrow pre(\lambda_1)))$. Conjoining this with (B.2.7) and (B.2.8) we get $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all $k, i \leq k \leq j$: $\sigma[..k] = I_{PO(\lambda)}$ holds. Since $PO(\lambda)$ is self consistent (because $(A \land post(\lambda_1)) \Rightarrow pre(\lambda_2)$ and both $PO(\lambda_1)$ and $PO(\lambda_2)$ are self consistent), we have $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda) \rrbracket$. Thus, by definition (4.6) we conclude that $\langle \sigma, i, j \rangle \in \llbracket \square A \rightarrow PO(\lambda) \rrbracket$.

Case 2.3: Assume that there exists $n, i \leq n \leq j$, such that:

\begin{align*}
\langle \sigma, i, n \rangle & \in \llbracket \lambda_1 \rrbracket \quad \text{for all } k, 1 \leq k < n: \sigma[k] \not\models (in(\lambda_2) \lor after(\lambda_2)) \quad (B.2.9) \\
\langle \sigma, n, j \rangle & \in \llbracket \lambda_2 \rrbracket \quad \text{for all } k, n < k \leq i: \sigma[k] \not\models (in(\lambda_1) \lor after(\lambda_1)) \quad (B.2.10) \\
\sigma[n] & = (after(\lambda_1) \land at(\lambda_2)) \quad \text{for all } k, 1 \leq k < n: \sigma[k] \not\models (in(\lambda_2) \lor after(\lambda_2)) \quad (B.2.11) \\
\sigma[n] & = (after(\lambda_1) \land at(\lambda_2)) \quad \text{for all } k, n < k \leq i: \sigma[k] \not\models (in(\lambda_1) \lor after(\lambda_1)) \quad (B.2.12)
\end{align*}

If $\sigma[..i] \not\models I_{PO(\lambda)}$ then, by definition, $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda) \rrbracket$. By definition (4.6), we conclude that $\langle \sigma, i, j \rangle \in \llbracket \square A \rightarrow PO(\lambda) \rrbracket$.

Now suppose $\sigma[..i] = I_{PO(\lambda)}$ holds.

First observe that due to Control Predicate Axiom CP4(a) and CP4(c) of Appendix A we have $at(\lambda) = at(\lambda_1)$ and $after(\lambda) = after(\lambda_2)$. Therefore, choosing $pre(\lambda_1)$ for $pre(\lambda)$ and choosing $post(\lambda_2)$ for $post(\lambda)$ we have:
\[ I_{PO(\alpha_1)} = (I_{PO(\alpha_1)} \land (at(\lambda) \Rightarrow pre(\lambda_1))) \] (B.2.14)

\[ I_{PO(\alpha_2)} = (I_{PO(\alpha_2)} \land (after(\lambda) \Rightarrow post(\lambda_2))) \] (B.2.15)

Because \( I_{PO(\alpha)} \) implies \( I_{PO(\alpha_1)} \), \( \sigma[..i] = I_{PO(\alpha_1)} \) holds. From (B.2.9) and (B.2.1) we have \( \langle \sigma, i, n \rangle \in \llbracket \square A \rightarrow PO(\lambda_1) \rrbracket \). Given the assumption (of Case 2) that \( \langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket \) we conclude that \( \langle \sigma, i, n \rangle \in \llbracket PO(\lambda_1) \rrbracket \).

From \( \langle \sigma, i, n \rangle \in \llbracket PO(\lambda_1) \rrbracket \) and assumption (above) \( \sigma[..i] = I_{PO(\alpha_1)} \) we have:

For all \( k, i \leq k < n \): \( \sigma[..k] = I_{PO(\alpha_1)} \)

(B.2.16)

By narrowing the range, we get:

For all \( k, i \leq k < n \): \( \sigma[..k] = I_{PO(\alpha_1)} \)

And, from (B.2.14) we get

For all \( k, i \leq k < n \): \( \sigma[..k] = I_{PO(\alpha_1)} \land (at(\lambda) \Rightarrow pre(\lambda_1)) \)

According to (B.2.12), \( I_{PO(\alpha_2)} \) is trivially true in states \( \sigma[..k] \) where \( i \leq k < n \). Moreover, from (B.2.15), \( I_{PO(\alpha_2)} \land (after(\lambda) \Rightarrow post(\lambda_2)) \) is also true in those states:

For all \( k, i \leq k < n \): \( \sigma[..k] = I_{PO(\alpha_1)} \land (at(\lambda) \Rightarrow pre(\lambda_1)) \land I_{PO(\alpha_2)} \land (after(\lambda) \Rightarrow post(\lambda_2)) \)

Equivalently, we have for all \( k, i \leq k < n \): \( \sigma[..k] = I_{PO(\alpha)} \).

From (B.2.16), again by narrowing the range, we conclude \( \sigma[..n] = I_{PO(\alpha_1)} \). By definition, \( I_{PO(\alpha_1)} \) implies \( after(\lambda_1) \Rightarrow post(\lambda_1) \), so by conjoining (B.2.11), we infer

\( \sigma[..n] = I_{PO(\alpha_1)} \land after(\lambda_1) \land post(\lambda_1) \land at(\lambda_2) \)

And, because of the assumption (Case 2) that \( \langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket \), we have \( \sigma[..n] = A \).

Thus we conclude:

\( \sigma[..n] = A \land I_{PO(\alpha_1)} \land after(\lambda_1) \land post(\lambda_1) \land at(\lambda_2) \)

Using (B.2.3) we get:

\( \sigma[..n] = I_{PO(\alpha_1)} \land after(\lambda_1) \land post(\lambda_1) \land at(\lambda_2) \land pre(\lambda_2) \)

\( PO(\lambda_2) \) is self consistent, so \( at(\lambda_2) \land pre(\lambda_2) \) implies \( I_{PO(\alpha_2)} \) and we have:

\( \sigma[..n] = I_{PO(\alpha_1)} \land I_{PO(\alpha_2)} \)

Using (B.2.14) and (B.2.15), we get

\( \sigma[..n] = I_{PO(\alpha_1)} \land (at(\lambda) \Rightarrow pre(\lambda_1)) \land I_{PO(\alpha_2)} \land (after(\lambda) \Rightarrow post(\lambda_2)) \)

or equivalently \( \sigma[..n] = I_{PO(\alpha)} \).

From (B.2.10) and (B.2.1) we have \( \langle \sigma, n, j \rangle \in \llbracket \square A \rightarrow PO(\lambda_2) \rrbracket \). Given the assumption (of Case 2) that \( \langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket \) we conclude that \( \langle \sigma, n, j \rangle \in \llbracket PO(\lambda_2) \rrbracket \).

From \( \langle \sigma, n, j \rangle \in \llbracket PO(\lambda_2) \rrbracket \) and \( \sigma[..n] = I_{PO(\alpha)} \) we have:
For all $k$, $n \leq k \leq j$: $\sigma[..k] = I_{PO}(\lambda_2)$

And, from (B.2.15) we get

For all $k$, $n \leq k \leq j$: $\sigma[..k] = I_{PO}(\lambda_2) \land (after(\lambda) \Rightarrow post(\lambda_2))$

According to (B.2.13), $I_{PO}(\lambda_2)$ is trivially true in traces $\sigma[..k]$ where $n \leq k \leq j$. Moreover, from (B.2.14), $I_{PO}(\lambda_2) \land (at(\lambda) \Rightarrow pre(\lambda_1))$ is also true in those traces:

For all $k$, $n \leq k \leq j$: $\sigma[..k] = (I_{PO}(\lambda_2) \land (at(\lambda) \Rightarrow pre(\lambda_1)) \land$

$I_{PO}(\lambda_2) \land (after(\lambda) \Rightarrow post(\lambda_2)))$,

or equivalently for all $k$, $n \leq k \leq j$: $\sigma[..k] = I_{PO}(\lambda_2)$.

Having proved for all $k$, $i \leq k < n$: $\sigma[..k] = I_{PO}(\lambda)$ and for all $k$, $n \leq k \leq j$: $\sigma[..k] = I_{PO}(\lambda)$ we conclude for all $k$, $i \leq k \leq j$: $\sigma[..k] = I_{PO}(\lambda)$. This means that $\langle \sigma, i, j \rangle \in \square A \rightarrow PO(\lambda)$ holds.  \qed
Lemma (Soundness of Cnstr-ParComp): The rule

\[ \square A \rightarrow PO(\lambda_1), \quad \square A \rightarrow PO(\lambda_2), \]
\[ \square A \rightarrow PO(\lambda_1) \text{ and } \square A \rightarrow PO(\lambda_2) \text{ are interference free} \]
\[ \square A \rightarrow \{pre(\lambda_1) \wedge pre(\lambda_2)\} \quad \lambda: \{PO(\lambda_1) \parallel PO(\lambda_2)\} \quad \{post(\lambda_1) \wedge post(\lambda_2)\} \]

is sound.

Proof. Assume that the three hypotheses are valid. Therefore, we have that \( PO(\lambda_1) \) is self-consistent, \( PO(\lambda_2) \) is self-consistent, and:

\[ \llbracket \lambda_1 \rrbracket \subseteq (\llbracket PO(\lambda_1) \rrbracket \cup \llbracket \square A \rrbracket) \] \hspace{1cm} (B.3.1)
\[ \llbracket \lambda_2 \rrbracket \subseteq (\llbracket PO(\lambda_2) \rrbracket \cup \llbracket \square A \rrbracket) \] \hspace{1cm} (B.3.2)

For all \( \lambda_\alpha \in Assig(\lambda_1) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [x := E] \):

\[ \llbracket \lambda_\alpha \rrbracket \subseteq (\llbracket \{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\} \lambda_\alpha: [x := E] \{I_{PO(\lambda_3)}\} \rrbracket \cup \llbracket \square A \rrbracket) \] \hspace{1cm} (B.3.3)

For all \( \lambda_\alpha \in Assig(\lambda_2) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [x := E] \):

\[ \llbracket \lambda_\alpha \rrbracket \subseteq (\llbracket \{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\} \lambda_\alpha: [x := E] \{I_{PO(\lambda_3)}\} \rrbracket \cup \llbracket \square A \rrbracket) \] \hspace{1cm} (B.3.4)

To establish validity of the rule’s conclusion, we must prove that \( \llbracket \lambda \rrbracket \subseteq \llbracket \square A \rightarrow PO(\lambda) \rrbracket \), where \( PO(\lambda) \) is \( \{pre(\lambda_1) \wedge pre(\lambda_2)\} \lambda: \{PO(\lambda_1) \parallel PO(\lambda_2)\} \{post(\lambda_1) \wedge post(\lambda_2)\} \). We do this by proving \( (\sigma, i, j) \in \llbracket \lambda \rrbracket \) implies \( (\sigma, i, j) \in \llbracket PO(\lambda) \rrbracket \cup \llbracket \square A \rrbracket \).

Let \( (\sigma, i, j) \in \llbracket \lambda \rrbracket \) hold.

Case 1: Assume \( (\sigma, i, j) \in \llbracket \square A \rrbracket \). According to definition (4.6) of a constrained proof outline, \( (\sigma, i, j) \in \llbracket \square A \rightarrow PO(\lambda) \rrbracket \) holds.

Case 2: Assume \( (\sigma, i, j) \in \llbracket \square A \rrbracket \). In order to conclude \( (\sigma, i, j) \in \llbracket \square A \rightarrow PO(\lambda) \rrbracket \) holds, we must show that \( (\sigma, i, j) \in \llbracket PO(\lambda) \rrbracket \) holds. We consider two cases.

 Case 2.1: Assume \( \sigma[.i] \neq I_{PO(\lambda)} \). According to definition (4.4) of the property defined by \( PO(\lambda) \), we conclude \( (\sigma, i, j) \in \llbracket PO(\lambda) \rrbracket \) holds.

 Case 2.2: Assume \( \sigma[.i] = I_{PO(\lambda)} \). We prove

For all \( k, i \leq k \leq j: \sigma[.k] = I_{PO(\lambda)} \)

by induction. This establishes that \( (\sigma, i, j) \in \llbracket PO(\lambda) \rrbracket \) holds, due to definition (4.4) of the property defined by \( PO(\lambda) \).

For the induction hypothesis we use

\[ P(h): i \leq h \quad \wedge \quad (h \leq j \Rightarrow \text{for all } k, i \leq k \leq h: \sigma[.k] = I_{PO(\lambda)}) \]

Base Case: Prove \( P(i) \). \( P(i) \) holds because it is implied by the assumption of (this) Case 2.2.

Induction Case: Prove \( P(h) \Rightarrow P(h+1) \). We assume that \( P(h) \) holds and prove \( P(h+1) \). If \( j \leq h \) then \( P(h+1) \) is trivially valid, so \( P(h) \Rightarrow P(h+1) \) is proved. We now consider the case
where $h < j$.

According to the definition (4.2) of $\llbracket \lambda \rrbracket$, we have $(\sigma[h], \sigma[h+1]) \in R_{\lambda^\prime}: [x := E]$ for some $\lambda^\prime \in \langle \text{Assig}(\lambda_1) \cup \text{Assig}(\lambda_2) \rangle$. Without loss of generality, suppose $\lambda^\prime \in \text{Assig}(\lambda_1)$ holds. Thus, we have $\langle \sigma, h, h+1 \rangle \in \llbracket \lambda^\prime \rrbracket$ and $(\sigma, h, h+1) \in \llbracket \lambda_1 \rrbracket$.

Given assumption $\langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket$ of (this) Case 2 and (B.3.1), we conclude $\langle \sigma, h, h+1 \rangle \in \llbracket PO(\lambda_1) \rrbracket$ holds. From assumptions $P(h)$ and $h < j$ we have $\sigma[. . . h] = I_{PO}(\lambda_1)$. Thus, $\sigma[. . . h] = I_{PO}(\lambda_1)$ because $I_{PO}(\lambda_1)$ implies $I_{PO}(\lambda_1)$ by definition:

\begin{align}
I_{PO}(\lambda_1): & \quad I_{PO}(\lambda_1) \land I_{PO}(\lambda_2) \land \\
& \quad (at(\lambda) \Rightarrow (pre(\lambda_1) \land pre(\lambda_2)) \land \\
& \quad (after(\lambda) \Rightarrow (post(\lambda_1) \land post(\lambda_2)))
\end{align}

(B.3.5)

Since $\sigma[. . . h] = I_{PO}(\lambda_1)$, according to definition (4.4) of $\llbracket PO(\lambda_1) \rrbracket$ we have that for all $k, i \leq k \leq j$ $\sigma[. . . k] = I_{PO}(\lambda_1)$ holds, because we know $\langle \sigma, i, j \rangle \in \llbracket PO(\lambda_1) \rrbracket$. And, since $i \leq h < j$ we conclude

$\sigma[. . . h+1] = I_{PO}(\lambda_1)$.

(B.3.6)

Given $\langle \sigma, h, h+1 \rangle \in \llbracket \lambda^\prime \rrbracket$, we conclude from (B.3.3) that:

$\langle \sigma, h, h+1 \rangle \in \llbracket \{ at(\lambda^\prime) \land I_{PO}(\lambda_1) \land I_{PO}(\lambda_2) \} \land \llbracket \square A \rrbracket \rrbracket$

Thus, from the assumption $\langle \sigma, i, j \rangle \in \llbracket \square A \rrbracket$ of (this) Case 2, we obtain:

$\langle \sigma, h, h+1 \rangle \in \llbracket \{ at(\lambda^\prime) \land I_{PO}(\lambda_1) \land I_{PO}(\lambda_2) \} \land \llbracket \square A \rrbracket \rrbracket$

(B.3.7)

From assumptions $P(h)$ and $h < j$ we have $\sigma[. . . h] = I_{PO}(\lambda_1)$ so, from (B.3.5), we conclude

$\sigma[. . . h] = I_{PO}(\lambda_1) \land I_{PO}(\lambda_2)$.

(B.3.8)

Using definition (4.4) of the property defined by a proof outline, we conclude from (B.3.7) that either $\sigma[. . . h] \neq I_{PO}(\lambda_1)$ or else

For all $k, h \leq k \leq h+1$ $\sigma[. . . k] = I_{PO}(\lambda_1)$

(B.3.9)

where

$I_{PO}(\lambda_2): (at(\lambda^\prime) \Rightarrow (at(\lambda^\prime) \land I_{PO}(\lambda_1) \land I_{PO}(\lambda_2))) \land (after(\lambda^\prime) \Rightarrow I_{PO}(\lambda_2))$

Definition (4.1) and $(\sigma[h], \sigma[h+1]) \in R_{\lambda^\prime}: [x := E]$ implies $\sigma[h] = at(\lambda^\prime)$. Conjoining $\sigma[h] = at(\lambda^\prime)$ and (B.3.8) allows us to rule out $\sigma[. . . h] \neq I_{PO}(\lambda_2)$ so (B.3.9) must hold.

From $(\sigma[h], \sigma[h+1]) \in R_{\lambda^\prime}: [x := E]$ and definition (4.1), we have $\sigma[h+1] = after(\lambda^\prime)$. We, therefore, conclude from (B.3.9) that

$\sigma[. . . h+1] = I_{PO}(\lambda_2)$

(B.3.10)

Finally, observe that, by construction, $I_{PO}(\lambda_1) \land I_{PO}(\lambda_2)$ is equivalent to $I_{PO}(\lambda)$ defined in (B.3.5). Thus, proving (B.3.6) and (B.3.10)—as we have—suffices for proving $\sigma[. . . h+1] = I_{PO}(\lambda)$, and $P(h+1)$ is proved. \qed
Lemma (Soundness of Cnstr-Equiv): The rule

\[
\text{Cnstr-Equiv: } \quad \Box A \rightarrow PO(\lambda), A \Rightarrow (l_{PO(\lambda)}=l_{PO'(\lambda)}), \text{ } PO'(\lambda) \text{ is self consistent} \\
\hline
\Box A \rightarrow PO'(\lambda)
\]

is sound.

Proof. Assume that the three hypotheses are valid. We prove that the rule's conclusion is valid by showing \( \llbracket \lambda \rrbracket \subseteq \llbracket \Box A \rightarrow PO'(\lambda) \rrbracket \).

Let \( \langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket \). From the validity of \( \Box A \rightarrow PO(\lambda) \), we have \( \llbracket \lambda \rrbracket \subseteq \llbracket \Box A \rightarrow PO(\lambda) \rrbracket \). We consider two cases.

Case 1: Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). According to definition (4.6) for the property defined by a constrained proof outline, we have \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow PO'(\lambda) \rrbracket \).

Case 2: Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). This means

For all \( k, i \leq k \leq j \): \( \sigma[.k]=A \) \hspace{1cm} (B.4.1)

Assuming \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \) also implies \( \langle \sigma, i, j \rangle \in \llbracket PO(\lambda) \rrbracket \), due to definition (4.6) of the property defined by a constrained proof outline. From definition (4.4) of \( \llbracket PO(\lambda) \rrbracket \) we conclude \( \sigma[.i] \neq I_{PO(\lambda)} \) or else for all \( k, i \leq k \leq j \): \( \sigma[.k]=I_{PO(\lambda)} \). By conjoining (B.4.1) with these, we infer \( \sigma[.i](A \land I_{PO(\lambda)}) \) or else for all \( k, i \leq k \leq j \): \( \sigma[.k]=(A \land I_{PO(\lambda)}) \).

The second hypothesis of Cnstr-Equiv implies that \( A \land I_{PO(\lambda)} \) equals \( A \land I_{PO'(\lambda)} \). Thus, we conclude \( \sigma[.i](A \land I_{PO'(\lambda)}) \) or else for all \( k, i \leq k \leq j \): \( \sigma[.k]=(A \land I_{PO'(\lambda)}) \) must hold. Given (B.4.1), if \( \sigma[.i](A \land I_{PO'(\lambda)}) \) holds then \( \sigma[.i] \neq I_{PO'(\lambda)} \) must hold. And, in that case, \( \langle \sigma, i, j \rangle \in \llbracket PO'(\lambda) \rrbracket \) according to definition (4.4) applied to \( \llbracket PO'(\lambda) \rrbracket \) because, by hypothesis, \( PO'(\lambda) \) is self consistent. In the case where for all \( k, i \leq k \leq j \): \( \sigma[.k]=(A \land I_{PO'(\lambda)}) \) holds, we conclude that for all \( k, i \leq k \leq j \): \( \sigma[.k]=I_{PO'(\lambda)} \) must hold, because \( (A \land I_{PO'(\lambda)}) \Rightarrow I_{PO'(\lambda)} \). Again, \( \langle \sigma, i, j \rangle \in \llbracket PO'(\lambda) \rrbracket \) according to definition (4.4) applied to \( \llbracket PO'(\lambda) \rrbracket \) because, by hypothesis, \( PO'(\lambda) \) is self consistent.

Having proved \( \langle \sigma, i, j \rangle \in \llbracket PO'(\lambda) \rrbracket \), we conclude \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow PO'(\lambda) \rrbracket \) based on definition (4.6) for the property defined by a constrained proof outline. \( \Box \)
Lemma (Relative Completeness of Cnstr-Assig): The rule

\[
\text{Cnstr-Assig: } \quad \frac{\{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \quad \Box A \rightarrow \{ p \} \lambda: [x := E] \quad \{ q \}}
\]

is relatively complete.

Proof. Assume that conclusion \( \Box A \rightarrow \{ p \} \lambda: [x := E] \quad \{ q \} \) is valid. We show that hypothesis \( \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \) is valid as well, by showing that if \( \langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket \) holds, then \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \) holds.

Let \( \langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket \) hold. We consider two cases.

Case 1: Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). According to the definition of \( \llbracket \Box A \rrbracket \):

\[
\text{For all } k, \quad i \leq k \leq j: \quad \sigma[.k]=A \tag{B.5.1}
\]

Using this and the assumption that \( \Box A \rightarrow \{ p \} \lambda: [x := E] \quad \{ q \} \) is valid, we conclude that \( \langle \sigma, i, j \rangle \in \llbracket \text{PO}(\lambda) \rrbracket \) and due to (4.4) one of the following holds:

\[
\sigma[.i] \neq \text{I}_{\text{PO}(\lambda)} \tag{B.5.2}
\]

\[
\text{for all } k, \quad i \leq k \leq j: \quad \sigma[.k]=\text{I}_{\text{PO}(\lambda)} \tag{B.5.3}
\]

where \( \text{I}_{\text{PO}(\lambda)}: (at(\lambda) \Rightarrow p) \land (after(\lambda) \Rightarrow q) \)

It suffices to prove that if (B.5.2) holds or (B.5.3) holds, then \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \).

Case 1.1: Assume (B.5.2) holds. Thus, \( \sigma[.i] \neq (at(\lambda) \Rightarrow p) \land (after(\lambda) \Rightarrow q) \) holds. Given (B.5.1), this implies \( \sigma[.i] = ((at(\lambda) \Rightarrow p \land \neg A) \lor (after(\lambda) \Rightarrow q \lor \neg A)) \) holds. This means \( \sigma[.i] \neq ((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A)) \), holds so, by definition, \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \) holds.

Case 1.2: Assume (B.5.3) holds. Given (B.5.1), this implies that

\[
\text{for all } k, \quad i \leq k \leq j: \quad \sigma[.k]=((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A))
\]

holds. Thus, by definition, \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \) holds.

Case 2: Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). According to the definition of \( \llbracket \Box A \rrbracket \):

\[
\text{Exists } k, \quad i \leq k \leq j: \quad \sigma[.k]=\neg A \tag{B.5.4}
\]

We consider three cases, according to the definition of \( \llbracket \lambda \rrbracket \) given in (4.2). Since \( \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \) is self consistent, it suffices to prove that \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \) holds.

Case 2.1: Assume \( j=i \) and \( \sigma[.i]=at(\lambda) \land \neg after(\lambda) \). By (B.5.4), we conclude \( \sigma[.i] \neq (at(\lambda) \Rightarrow p \land A) \), so \( \sigma[.i] \neq ((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A)) \). Thus, by definition, \( \langle \sigma, i, j \rangle \in \llbracket \{ p \land A \} \lambda: [x := E] \quad \{ q \lor \neg A \} \rrbracket \) holds.

Case 2.2: Assume \( j=i \) and \( \sigma[.i]=\neg at(\lambda) \land after(\lambda) \). By (B.5.4), we conclude \( \sigma[.i] \neq (after(\lambda) \Rightarrow q \lor \neg A) \), so \( \sigma[.i] \neq ((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A)) \). Since \( i=j \) holds, so does

\[
\text{for all } k, \quad i \leq k \leq j: \quad \sigma[.k]=((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A))
\]
Thus, by definition, $\langle \sigma, i, j \rangle \in \llbracket P \land A \rrbracket \lambda: [x := E] \{q \lor \neg A\}$ holds.

Case 2.3: Assume $j = i + 1$ and $\sigma[.i] = at(\lambda) \land \neg after(\lambda)$. If $\sigma[.i] = (\neg p \lor \neg A)$ then $\sigma[.i] \models ((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A))$ and, by definition, $\langle \sigma, i, j \rangle \in \llbracket P \land A \rrbracket \lambda: [x := E] \{q \lor \neg A\}$ holds. If, on the other hand, $\sigma[.i] = (p \land A)$, then we conclude from $j = i + 1$ and (B.5.4) that $\sigma[.j] = (q \lor \neg A)$ holds, so

for all $k$, $i \leq k \leq j$: $\sigma[.k] = ((at(\lambda) \Rightarrow p \land A) \land (after(\lambda) \Rightarrow q \lor \neg A))$

also holds. Again, $\langle \sigma, i, j \rangle \in \llbracket P \land A \rrbracket \lambda: [x := E] \{q \lor \neg A\}$ holds. \qed
Lemma (Relative Completeness of Cnstr-SeqComp): Assume that all valid constrained proof outlines $\square A \rightarrow PO^*(\lambda_1)$ and $\square A \rightarrow PO^*(\lambda_2)$ are provable, and assume that all valid predicate logic formulas are provable. Then

$$\square A \rightarrow \{ p \} \lambda: [PO(\lambda_1); PO(\lambda_2)] \{ q \}$$  \hspace{1cm} (B.6.1)

is provable if valid.

Proof. Assume (B.6.1) is valid. Let

$$\square A \rightarrow \{ I \} \lambda: [PO'(\lambda_1); PO'(\lambda_2)] \{ I \}$$  \hspace{1cm} (B.6.2)

be a proof outline for $\lambda: [\lambda_1; \lambda_2]$ in which every assertion in $PO'(\lambda_1)$ and $PO'(\lambda_2)$ is

$$I: I_{PO(\lambda_1)} \land I_{PO(\lambda_2)} \land (at(\lambda) \Rightarrow p) \land (after(\lambda) \Rightarrow q).$$

Observe that (B.6.2) is valid by Cnstr-Equiv because the proof outline invariant for (B.6.1) equals $I$ and (B.6.1) is valid.

We show below that

$$\square A \rightarrow \{ I \} PO'(\lambda_1) \{ I \}$$  \hspace{1cm} (B.6.3)

$$\square A \rightarrow \{ I \} PO'(\lambda_2) \{ I \}$$  \hspace{1cm} (B.6.4)

are both valid. This implies that (B.6.3) and (B.6.4) are provable by the assumptions of this lemma. By construction, $post(\lambda_1)$ and $pre(\lambda_2)$ in (B.6.2) are both $I$, so $A \land post(\lambda_1) \Rightarrow pre(\lambda_2)$ is valid and, by the lemma's assumption, provable. The three hypotheses needed for using Cnstr-SeqComp to deduce (B.6.2) are now discharged. We can thus use Cnstr-Equiv to deduce (B.6.1)

We prove that (B.6.3) is valid by showing that $[\lambda_1] \subseteq [\square A \rightarrow \{ I \} PO'(\lambda_1) \{ I \}]$ holds. Let $\langle \sigma, i, j \rangle \in [\lambda_1]$ hold. This means that $\langle \sigma, i, j \rangle \in [\lambda]$ holds as well. We conclude

$$\langle \sigma, i, j \rangle \in [\square A \rightarrow \{ I \} \lambda: [PO'(\lambda_1); PO'(\lambda_2)] \{ I \}]$$  \hspace{1cm} (B.6.5)

because (B.6.2) was proved valid above.

Case 1: Assume $\langle \sigma, i, j \rangle \in [\square A]$. According to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\square A \rightarrow \{ I \} PO'(\lambda_1) \{ I \}]$.

Case 2: Assume $\langle \sigma, i, j \rangle \notin [\square A]$. This assumption and (B.6.5) imply $\langle \sigma, i, j \rangle \in [\{ I \} \lambda: [PO'(\lambda_1); PO'(\lambda_2)] \{ I \}]$.

The proof outline invariant for (B.6.2) is

$$I': \lambda' \in Lab(\lambda) ((at(\lambda') \Rightarrow I) \land (after(\lambda') \Rightarrow I))$$  \hspace{1cm} (B.6.6)

where $cp$ ranges over the control predicates of $\lambda$. From definition (4.4) of $[PO(\lambda)]$ we infer that $\sigma[.i]|I'$ or else for all $k, i \leq k \leq j: \sigma[.k]=I'$.

Case 2.1: Assume $\sigma[.i]|I'$. Thus, by definition, $\sigma[.i]=I'$ holds. According to definition (B.6.6) for $I'$, we conclude $\sigma[.i]=((\forall cp) \land \neg I)$ holds, which implies that $\sigma[.i]|I$ holds. By definition (4.4) of $[PO(\lambda_1)]$, we conclude $\langle \sigma, i, j \rangle \in [\{ I \} PO'(\lambda_1) \{ I \}]$. And, according to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\square A \rightarrow \{ I \} PO'(\lambda_1) \{ I \}]$.
Case 2.2: Assume for all $k, i \leq k \leq j$: $\sigma[..k] \models I'$. Since $\langle \sigma, i, j \rangle \in \llbracket \lambda_1 \rrbracket$ we have for all $k, i \leq k \leq j$: $\sigma[..k] \models (in(\lambda_1) \lor after(\lambda_1))$. By construction $(I' \land (in(\lambda_1) \lor after(\lambda_1))) \Rightarrow I$ is valid. So we infer that for all $k, i \leq k \leq j$: $\sigma[..k] \models I$. $I$ implies $I_{PO(\lambda_i)}$, and this allows us to conclude for all $k, i \leq k \leq j$: $\sigma[..k] \models I_{PO(\lambda_i)}$. Thus, by definition (4.4) of $\llbracket PO(\lambda_1) \rrbracket$, we conclude $\langle \sigma, i, j \rangle \in \llbracket \{I\} PO'(\lambda_1) \{I\} \rrbracket$. According to definition (4.6) for the property defined by a constrained proof outline, we conclude $\langle \sigma, i, j \rangle \in \llbracket \square A \rightarrow \{I\} PO'(\lambda_1) \{I\} \rrbracket$.

A similar argument establishes that (B.6.4) is valid.
Lemma (Relative Completeness of Cnstr-ParComp): Assume that all valid constrained proof outlines \( \Box A \to PO^*(\lambda_1) \) and \( \Box A \to PO^*(\lambda_2) \) are provable, and assume that all valid constrained proof outlines involving the individual assignment statements in \( \lambda_1 \) and \( \lambda_2 \) are provable. Then

\[
\Box A \to \{pre(\lambda_1) \land pre(\lambda_2)\} \land \{PO(\lambda_1) \lor PO(\lambda_2)\} \land \{post(\lambda_1) \land post(\lambda_2)\} \tag{B.7.1}
\]
is provable if valid.

Proof. Assume (B.7.1) is valid. Let

\[
\Box A \to \{I\} \land \{PO'(\lambda_1) \lor PO'(\lambda_2)\} \tag{B.7.2}
\]
be a proof outline for \( \lambda: [\lambda_1 \lor \lambda_2] \) in which every assertion in \( PO'(\lambda_1) \) and \( PO'(\lambda_2) \) is

\[
I: I_{PO(\alpha_1)} \land I_{PO(\alpha_2)} \land (at(\lambda) \Rightarrow (pre(\lambda_1) \land pre(\lambda_2))) \land (after(\lambda) \Rightarrow (post(\lambda_1) \land post(\lambda_2))).
\]

Observe that (B.7.2) is valid by Cnstr-Equiv, because the proof outline invariant for (B.7.1) equals \( I \) and (B.7.1) is valid.

We show below that

\[
\Box A \to \{I\} \land PO'(\lambda_1) \tag{B.7.3}
\]
\[
\Box A \to \{I\} \land PO'(\lambda_2) \tag{B.7.4}
\]
are both valid. This implies that (B.7.3) and (B.7.4) are provable by the assumptions of this lemma. We also show:

For all \( \lambda_\alpha \in Assig(\lambda_2) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [x := E] \):

\[
\Box A \to \{at(\lambda) \land I_{PO(\alpha_2)} \land I_{PO(\alpha_1)}\} \land \lambda_\alpha: [x := E] \land \{I_{PO(\alpha_1)}\} \tag{B.7.5}
\]
is valid.

For all \( \lambda_\alpha \in Assig(\lambda_1) \), where \( \lambda_\alpha \) is the assignment \( \lambda_\alpha: [x := E] \):

\[
\Box A \to \{at(\lambda) \land I_{PO(\alpha_1)} \land I_{PO(\alpha_2)}\} \land \lambda_\alpha: [x := E] \land \{I_{PO(\alpha_2)}\} \tag{B.7.6}
\]
is valid.

By the lemma’s assumption, interference-freedom obligations (B.7.4) and (B.7.5) are thus provable. The three hypotheses needed for using Cnstr-ParComp to deduce (B.7.2) are now discharged. We can thus use Cnstr-Equiv to deduce (B.7.1).

We prove that (B.7.3) is valid by showing that \( \llbracket \lambda_1 \rrbracket \subseteq \llbracket \Box A \to \{I\} PO'(\lambda_1) \{I\} \rrbracket \) holds. Let \( \langle \sigma, i, j \rangle \in \llbracket \lambda_1 \rrbracket \) hold. This means that \( \langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket \) holds as well. We conclude

\[
\langle \sigma, i, j \rangle \in \llbracket \Box A \to \{I\} \land \{PO'(\lambda_1) \lor PO'(\lambda_2)\} \{I\} \rrbracket \tag{B.7.7}
\]
because (B.7.2) was proved valid above.

Case 1: Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). According to definition (4.6) for the property defined by a constrained proof outline, we have \( \langle \sigma, i, j \rangle \in \llbracket \Box A \to \{I\} PO'(\lambda_1) \{I\} \rrbracket \).

Case 2: Assume \( \langle \sigma, i, j \rangle \notin \llbracket \Box A \rrbracket \). This assumption and (B.7.7) imply \( \langle \sigma, i, j \rangle \in \llbracket \{I\} \land \{PO'(\lambda_1); PO'(\lambda_2)\} \{I\} \rrbracket \).
The proof outline invariant for (B.7.2) is

\[ I' \quad \land \quad (cp \Rightarrow I), \quad \text{(B.7.8)} \]

where \( cp \) ranges over the control predicates of \( \lambda \). From definition (4.4) of \( \llbracket PO(\lambda) \rrbracket \) we infer that \( \sigma[.i][1][1]' \) or else for all \( k, i \leq k \leq j : \sigma[.k][1][1]' \).

**Case 2.1:** Assume \( \sigma[.i][1][1]' \). Thus, by definition, \( \sigma[.i][1][1]' \land \lnot I' \) holds. According to definition (B.7.8) for \( I' \), we conclude \( \sigma[.i][1][1]' \land (\lnot cp \land \lnot I) \) holds, which implies that \( \sigma[.i][1][1]' \) holds. By definition (4.4) of \( \llbracket PO(\lambda_1) \rrbracket \), we conclude \( \langle \sigma, i, j \rangle \in \llbracket I \rrbracket \ \llbracket PO'(\lambda_1) \rrbracket \ \llbracket I \rrbracket \). And, according to definition (4.6) for the property defined by a constrained proof outline, we have \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow \{ I \} \ \llbracket PO'(\lambda_1) \rrbracket \ \llbracket I \rrbracket \).

**Case 2.2:** Assume for all \( k, i \leq k \leq j : \sigma[.k][1][1]' \). Since \( \langle \sigma, i, j \rangle \in \llbracket \lambda_1 \rrbracket \) we have for all \( k, i \leq k \leq j : \sigma[.k][1][1]' \Rightarrow (in(\lambda_1) \lor after(\lambda_1)) \). By construction \( I' \land (in(\lambda_1) \lor after(\lambda_1)) \Rightarrow I \) is valid. So we infer that for all \( k, i \leq k \leq j : \sigma[.k][1][1]' \Rightarrow I \) holds. By definition (4.4) of \( \llbracket PO(\lambda_1) \rrbracket \), we conclude \( \langle \sigma, i, j \rangle \in \llbracket \llbracket I \rrbracket \ \llbracket PO'(\lambda_1) \rrbracket \ \llbracket I \rrbracket \). And, according to definition (4.6) for the property defined by a constrained proof outline, we conclude \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow \{ I \} \ \llbracket PO'(\lambda_1) \rrbracket \ \llbracket I \rrbracket \).

A similar argument establishes that (B.7.4) is valid.

We prove that (B.7.5) is valid by showing that

\[ \llbracket \lambda_\alpha \rrbracket \subseteq \llbracket \Box A \rightarrow \{ at(\lambda) \land I_{P_0(\alpha_1)} \land I_{P_0(\alpha_2)} \} \ \lambda_\alpha : \{ x := E \} \ \{ I_{P_0(\alpha_2)} \} \rrbracket \]

holds for all \( \lambda_\alpha \in Assig(\lambda_1) \), where \( \lambda_\alpha \) is an assignment \( \lambda_\alpha : \{ x := E \} \).

For arbitrary \( \lambda_\alpha \), let \( \langle \sigma, i, j \rangle \in \llbracket \lambda_\alpha \rrbracket \) hold. This means that \( \langle \sigma, i, j \rangle \in \llbracket \lambda \rrbracket \) holds as well. We conclude

\[ \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow \{ I \} \ \lambda : \{ PO'(\lambda_1) \lor PO'(\lambda_2) \} \ \llbracket I \rrbracket \] (B.7.9)

because (B.7.2) was proved valid above. Define

\[ PO*(\lambda_\alpha) : \{ at(\lambda) \land I_{P_0(\alpha_1)} \land I_{P_0(\alpha_2)} \} \ \lambda_\alpha : \{ x := E \} \ \{ I_{P_0(\alpha_2)} \} \]

**Case 1:** Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). According to definition (4.6) for the property defined by a constrained proof outline, we have \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rightarrow PO*(\lambda_\alpha) \rrbracket \).

**Case 2:** Assume \( \langle \sigma, i, j \rangle \in \llbracket \Box A \rrbracket \). This assumption and (B.7.9) imply \( \langle \sigma, i, j \rangle \in \llbracket \{ I \} \ \lambda : \{ PO'(\lambda_1) \lor PO'(\lambda_2) \} \ \llbracket I \rrbracket \).

The proof outline invariant for (B.7.9) is

\[ I' : \land \quad (cp \Rightarrow I) \land (after(\lambda') \Rightarrow I) \] (B.7.10)

where \( cp \) ranges over the control predicates of \( \lambda \). From definition (4.4) of \( \llbracket PO(\lambda) \rrbracket \) we infer that \( \sigma[.i][1][1]' \) or else for all \( k, i \leq k \leq j : \sigma[.k][1][1]' \).

**Case 2.1:** Assume \( \sigma[.i][1][1]' \). Thus, by definition, \( \sigma[.i][1][1]' \land \lnot I' \) holds. According to definition (B.7.8) for \( I' \), we conclude \( \sigma[.i][1][1]' \land (\lnot cp \land \lnot I) \) holds, which implies that \( \sigma[.i][1][1]' \) holds. By definition (4.4) of \( \llbracket PO*(\lambda_\alpha) \rrbracket \), we conclude \( \langle \sigma, i, j \rangle \in \llbracket PO*(\lambda_\alpha) \rrbracket \). And, according to definition (4.6) for the property defined by a
constrained proof outline, we have $⟨σ, i, j⟩∈ [[ □A → PO^*(λ_α) ]]$.

**Case 2.2:** Assume for all $k, i ≤ k ≤ j$: $σ[.k]=I'$. Since $⟨σ, i, j⟩∈ [[ λ_α ]]$ holds we have

for all $k, i ≤ k ≤ j$: $σ[.k]=I(σ(λ_α) ∨ after(λ_α))$.

By construction $(I' ∨ (at(λ_α) ∨ after(λ_α))) ⇒ I'$ is valid. So we infer that

for all $k, i ≤ k ≤ j$: $σ[.k]=I$.

Since $I(λ_α) ⇒ (I_{PO}(λ_1) ∧ I_{PO}(λ_2)) ∨ (after(λ_α) ⇒ I_{PO}(λ_2))$ and this allows us to conclude

for all $k, i ≤ k ≤ j$: $σ[.k]=I_{PO^*(λ_α)}$. Thus, by definition (4.4) of $[[ PO^*(λ_α) ]]$, we conclude

$⟨σ, i, j⟩∈ [[ PO^*(λ_α) ]]$. According to definition (4.6) for the property defined by a constrained proof outline, we conclude $⟨σ, i, j⟩∈ [[ □A → PO^*(λ_α) ]]$.

A similar argument establishes that (B.7.6) is valid. □
Theorem (Relative Completeness): Cnstr-Assig, Cnstr-SeqComp and Const-ParComp comprise a (relatively) complete deductive system.

Proof. The proof is by structural induction on programs.

Base Case: A program consisting of a single assignment statement. This case then follows by the Relative Completeness of Cnstr-Assig Lemma.

Induction Case: This case then follows by the Relative Completeness of Cnstr-SeqComp and Relative Completeness of Cnstr-ParComp Lemmas.