

Calvinism: a static analysis for multi-threaded program verification

Abstract

As the number of businesses that rely on computer software grows the costs of bugs in computer code increases. At the same time an increase in the complexity of code makes it difficult and costly to verify that applications acts in the manner proscribed. By leveraging Hoare Logic, Calvinism provides a framework for static verification of a program. At the minor cost to the programmer of providing guard annotations, Calvinism Analysis can check that a program meets specifications.

1 Introduction

Software verification is no longer the abstract theoretical problem, but an important and pressing issue. The lack of adequate program testing not only costs the US economy 59.5 billion annually but can harm people if it results in the failure of mission-critical systems. To compound the problem the size and complexity of software is increasing making the conventional method of verification: software testing in conjunction with code coverage inadequate. The ideal checker would be static, easy to use, and expressive enough to capture the intent of the programmer. Though proofs that programs are true to their specifications can be written they are difficult to write and therefore aren't used for large scale commercial products. Automated theorem provers can verify properties of a program but often require extensive programmer aide to use. Though model checking could be applied the state space explosion in nondeterministic programs makes the approach untenable. Another approach, types, uses little or no support from programers however is limited in the properties it can check. Calvinism presents a theoretical framework for a static assertion checker which can statically prove many properties of a program that are expressible in assertions.

Adding to the complexity of problems has been the trouble of reasoning about programs with multiple threads. This nondeterminism causes many problems in verifying programs however threads have the ability to bring substantial improvements to the speed of programs. For this reason the struggle to provide verification for non-deterministic programs should be taken up. Calvinism presents a possible solution to verifying properties of multithreaded programs using the concept of atomicity through guarded variables and reduction couched in terms of Hoare Logic.

2 Background

Calvinism is draws much from its predecessor Calvin [2] and the paper on atomicity [1] written by its creators. The work on Calvin presented an implementation of a multithreaded assertion checker which worked off the rely-guarantee reasoning of Jones [4] to reason about multi-threaded programs modularly. The paper reasoned about each function separately by replacing function calls with simulations mimicking how each call would have changed the state. Calvin is able to do this by verifying, with an assert, the methods assumptions and inlining changes to the state given by the method's guarantees.

Calvin also leverages the idea of atomicity [1] to reason about multithreaded programs as if they were sequential. In order to regulate access to shared variables Calvin inserts assertions before the access occurs. Secondly Calvin reasons about the atomicity of a method in order to justify sending the program to a sequential checker. Atomicity occurs in methods whose actions can be modeled by serial program in spite of the actions of other threads. This can be reasoned using Liptons concept of reduction [5].

What this paper does is introduce a theoretical framework, based on Calvin, for creating a checker which directly accesses the validity assertions through a set of axioms. This type of analysis is simpler than that of Calvin's which can facilitate further research, provide a lucid view of the work's flaws and a clear proof of soundness.

3 Atomicity, Reduction, and Validity

The goal of this checker is to decide whether any assertions fail in the function. In order to simplify assertion checks to sequential reasoning, Calvinism requires that the method is atomic. The reasoning for atomicity is in turn aided by enforcement of the access predicates. One check is for obedience to rules of access predicates. These predicates guard shared variables and can model general synchronization mechanisms. Instead of creating a special mutex construct guarding a variable, c , an access predicate, $A(c)$ will act in a similar manner.

$$\begin{aligned} & \text{Variable } m, c; \\ & A(m) = (m == 0) \vee (m == tid) \\ & A(c) = A(m) \end{aligned}$$

Note that access guard are malleable and can encompass any access pattern describable by boolean logic using the state for variables. These guards are necessary, not to prevent race freedom but to expose the access patterns of shared variables to the checker. These guards become important when the checker needs to describe access shared variables. It may use a single mechanism to describe it whereas the programmer may use many different constructs from mutexes to reader-writer locks.

Calvinism checks for the atomicity of a method by using the notion of reduction first introduced by Lipton. The premise is that if two commands in a method can be considered one atomic, sequential command if one of the following holds: the post-condition of the first command implies exclusive access to the variables accessed by the command, or the precondition of the second command implies exclusive access to variables in the command. Exclusive access can be algorithmically determined from the access predicate to the variable.

$$E(v) = \forall x.x \neq tid \quad \neg A(v).$$

In the case of the above variable, c , $E(c) = (x == tid)$. For example if $\vdash_a \{P\}$ while m do $c := 0\{Q\}$ the checker would either need to show that $P \Rightarrow E(c)$ or $Q \Rightarrow E(c)$ to prove atomicity in the greater program.

Finally a Calvinism checker would determine whether a function is valid. Pre-condition and post-condition checks in the style of Hoare [3] are used in the axiomatic rules to determine whether asserts will fail. As Calvinism already fails if the function cannot be treated atomically, a checker based on Calvinism can reason about asserts of non-deterministic programs as if it were sequential. With pre-conditions and post-conditions a checker can algorithmically construct a proof that an assert will not occur given a precondition to the program.

4 Operational Semantics

To study the theorem checker we created a toy language to check which is small enough to reason about but expressive enough to capture the essential parts of a language. The language is displayed in Table 1.

$$s ::= \text{assume } b \mid \text{assert } b \mid \text{when } b \text{ do } v := e \mid s^* \mid s_1; s_2 \mid s_1 \square s_2 \mid s_1 \parallel s_2 \mid \text{wrong} \mid \text{skip}$$

Table 1 - The language syntax

Note that the \square symbol adequately models the familiar control structures such as if then and do while structures.

$$\begin{aligned} &\text{if } b \text{ then } s_1 \text{ else } s_2 \cong \text{assert } b; s_1 \square \text{assert } \neg b; s_2 \\ &\text{do } s \text{ while } b \cong s^*; \text{assert } \neg b \end{aligned}$$

The when/do construct acts like a test and set construct, blocking until an assertion is true and then atomically setting a value. The construct also acts as ordinary assignment by setting the guard b to true. Each thread is given a unique local variable tid which acts as an identifier. Depending on whether the thread is the left or right child of the thread invoking the forking construct, \parallel , the thread's tid is either $tid.0$ or $tid.1$ where tid is the parent thread's identifier. Each program begins with one thread with the $tid = \phi$. The following table displays key operational rules, for the complete semantics refer to the appendix.

$\frac{\langle s_1, \sigma \rangle \rightarrow_{tid} \langle s'_1, \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow_{tid} \langle s'_1; s_2, \sigma' \rangle}$	$\frac{\langle s_1, \sigma \rangle \rightarrow_{tid.0} \langle s'_1, \sigma' \rangle}{\langle s_1 \parallel s_2, \sigma \rangle \rightarrow_{tid} \langle s'_1 \parallel s_2, \sigma' \rangle}$	$\frac{\langle s_2, \sigma \rangle \rightarrow_{tid.1} \langle s'_2, \sigma' \rangle}{\langle s_1 \parallel s_2, \sigma \rangle \rightarrow_{tid} \langle s_1 \parallel s'_2, \sigma' \rangle}$
$\frac{\neg b(\sigma, tid)}{\langle \text{assert } b, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle}$	$\frac{b(\sigma) \quad e(\sigma, tid) = x}{\langle \text{when } b \text{ do } v := e, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, [x/v]\sigma \rangle}$	$\frac{}{\langle s^*, \sigma \rangle \rightarrow_{tid} \langle s; s^*, \sigma \rangle}$
$\frac{}{\langle s^*, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, \sigma \rangle}$	$\frac{}{\langle s_1 \square s_2, \sigma \rangle \rightarrow_{tid} \langle s_1, \sigma \rangle}$	$\frac{}{\langle s_1 \square s_2, \sigma \rangle \rightarrow_{tid} \langle s_2, \sigma \rangle}$

Table 2 - Selected operational rules¹

5 Axiomatic Semantics

The key to the axiomatic semantics of Calvinism is that to verify the assertions of multi-threaded programs it requires atomicity. To prove atomicity in turn the analysis must be able to rely on the access predicates to determine whether a method is atomic. So Calvinism checks for compliance to access predicates, atomicity using the access predicates and that assertions pass using reasoning about atomicity. Table 3 displays some key axiomatic rules.

In order to check guard predicates and atomicity along with the validity of a method, Calvinism incorporates additional preconditions into its axiomatic rules. A rule to simply detect the validity of the program $\text{assert } b$ would have the condition that the precondition, P , implies b . However in order to check that the program complies with access predicates the analysis also requires that P

¹ $\sigma : \text{Variable} \rightarrow \text{Values} \quad e : \text{State} \times \text{Threads} \rightarrow \text{Values} \quad b : \text{State} \times \text{Threads} \rightarrow \text{Boolean}$

implies A(b). (9.3) A similar check must be made for all commands with no subcommands ². (i.e. assume, assert, when/do, wrong, & skip)

For those commands with subcommands (i.e. $s_1; s_2, s_1 \parallel s_2, s^*$) the axiomatic semantics require that the command be atomic. For example a sequence of commands $s_1; s_2$ must not only have a middle state such that $\vdash_a \{P\} s_1 \{Q\}$ & $\vdash_a \{Q\} s_2 \{R\}$ to be valid but Q must also imply exclusive access to the variables in s_1 or in s_2 .

$$\frac{P \Rightarrow A(b) \quad P \Rightarrow b}{\vdash_a \{P\} \text{ assert } b \{P \wedge b\}} \qquad \frac{P \wedge b \Rightarrow A(\text{ when } b \text{ do } v := e) \quad P \wedge b \Rightarrow Q[e/v]}{\vdash_a \{P\} \text{ when } b \text{ do } v := e \{Q\}}$$

$$\frac{\vdash_a \{P\} s_1 \{Q\} \quad \vdash_a \{Q\} s_2 \{R\} \quad [Q \Rightarrow E(s_1) \vee Q \Rightarrow E(s_2)]}{\vdash_a \{P\} s_1; s_2 \{R\}} \qquad \frac{\vdash_a \{P\} s_1; s_2 \{Q\} \quad \vdash_a \{P\} s_2; s_1 \{Q\}}{\vdash_a \{P\} s_1 \parallel s_2 \{Q\}}$$

Table 3 - Selected axiomatic rules

6 Soundness

Given that the checker adheres to the axiomatic semantics can one guarantee that programs which pass the checker will never fail? This leads us to the Soundness Lemma.

$$\text{If } \vdash_a \{P\} s \{Q\} \text{ holds and } \sigma \models_{tid} P \\ \text{then } \langle s, \sigma \rangle \not\rightarrow_{tid}^* \text{ wrong}$$

This papers proof of soundness rests on a lemma that reducing a program which passes the checker will preserve the invariant that the program passes the checker. By induction on each program reduction (\rightarrow_{tid}), it follows that if there exists any program s' such that $\langle s, \sigma \rangle \rightarrow_{tid} \langle s', \sigma' \rangle$ then $\vdash_a \{P'\} s' \{Q\}$. As any valid program cannot be the program wrong the theorem will also holds. The proof is located in the appendix. It turns out that certain cases of the Lemma are trivial to prove. For example proving that reducing $s_1; s_2$ to $s'_1; s_2$ is merely a matter of gathering all the preconditions of the lemma to wield induction on the smaller case of s_1 reducing to s'_1 .

$$s = s_1; s_2 \quad s' = s'_1; s_2$$

by $\vdash_a \{P\} s_1; s_2 \{Q\}$ implies that $\vdash_a \{P\} s_1 \{Q\}$ [9.5]

$$\sigma \models_{tid} P$$

By induction it follows that $\exists P'. \vdash_a \{P'\} s'_1 \{Q\}$ & $\sigma' \models_{tid} P'$

Furthermore by 9.5 $\vdash_a \{P\} s_1; s_2 \{R\} \Rightarrow \vdash_a \{Q\} s_2 \{R\}$

so $\vdash_a \{P\} s'_1; s_2 \{R\}$ & $\sigma' \models_{tid} P'$

The proof is only partially complete but the case seems intuitively to hold and I hope that the case for the forking construct, \parallel , will be proven soon.

7 Limitations

The most important limitation on this work is that the meta language is restrictive and doesnt completely emulate all the important aspects of a language. Most egregious is its lack of function calls. This could be solved by simulating a function as the Calvin checker does. The programmer

²These type of commands are in fact what are commonly known as atomic commands. Just to make things confusing

would annotate each function with preconditions and postconditions. The cost of this is paid in programmer effort which in turn is directly proportional to acceptance.

A restriction that the checker imposes is that the method be atomic. This is restrictive in that many verifiable functions are not atomic. Take for example a busy wait which tries to acquire a lock, a shared variable, until it succeeds. As such a function would relinquish the lock on the variable to allow the holder to release the lock. While this function is not atomic (each attempt to acquire the lock is an atomic section) it certainly can should be permitted to pass the checker. One potential solution that has not been thoroughly examined is that an alternate rule for sequences should be also allowed:

$$\frac{\vdash_a \{P\}s_1\{Q\} \quad \vdash_a \{Q\}s_2\{R\} \quad [\forall x \in \alpha(Q) Q \Rightarrow E(x)]}{\vdash_a \{P\}s_1; s_2\{R\}}$$

By examining whether the intermediate state Q has exclusive access to its variables we can be sure that despite actions from other threads the variables which the next command relies on will remain solid. Adding such a rule would verify even more methods than a logic with the existing rule. Whether it would solve the problem of non-atomic valid methods completely is subject to further study.

8 Conclusion

The research which I did this semester helped me have a better grasp of axiomatic semantics. Whether or not this research will turn out to improve software verification will depend greatly on the development of the limitations to the system as they are fairly restrictive. Nonetheless this is a start to finding a clear axiomatic solution to multithreaded program verification.

References

- [1] Stephen N. Freund Cormac Flanagan and Shaz Qadeer. Exploiting purity for atomicity. *ACM SIGSOFT Software Engineering Notes*, 29(4):221–231, July 2004.
- [2] Stephen N. Freund and Shaz Qadeer. Checking Concise Specifications for Multithreaded Software. *Journal of Object Technology*, 3(6):81–101, June 2004.
- [3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
- [4] Cliff B. Jones. Specification and Design of (Parallel) Programs. *Proceedings of IFIP '83*, pages 321–332, 1983.
- [5] Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, December 1975.

9 Appendix

By convention:

$$\sigma \in State$$

$tid \in Threads (T_{id})$
 $v \in Variables$

Define:

$s ::= \text{assume } b \mid \text{assert } b \mid \text{when } b \text{ do } v := e \mid s^* \mid s_1; s_2 \mid s_1 \square s_2 \mid s_1 \parallel s_2 \mid \text{wrong} \mid \text{skip}$

$\sigma : Variable \rightarrow Values$

$e : State \times T_{id} \rightarrow Values$

$b : State \times T_{id} \rightarrow Boolean$

$P, Q : State \times T_{id} \rightarrow Boolean$

$b, P, Q(\sigma, tid)$ = predicates b, P or Q given σ and tid

b is reserved for booleans within the language.

P, Q are reserved for pre & post conditions in Axiomatic Semantics.

$\alpha(s)$ = variables accessed by the statement s

$A, E : Var \times State \times T_{id} \rightarrow Boolean$

$A(v, \sigma, tid)$ = the conjunction of access predicates guarding v given σ and tid

$A(s, \sigma, tid) = \forall x \in \alpha(s) A(x, \sigma, tid) = true$

$E(v, \sigma, tid) = \forall j \in T_{id} j \neq tid \Rightarrow \neg A(v, \sigma, j)$

$E(s, \sigma, tid) = \forall x \in \alpha(s) E(x, \sigma, tid)$

Calls to $e, b, P, Q, A(x), E(x)$ abbreviate $\forall \sigma, tid b(\sigma, tid); \forall \sigma, tid A(x, \sigma, tid)$, etc.

$A \vdash \{P\}s\{Q\} \equiv \vdash_a \{P\}s\{Q\}$

An empty program is by convention $\langle \text{skip}, \sigma \rangle$

9.1 Axiomatic Semantics

$$\frac{\vdash_a \{P\}s\{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\vdash_a \{P'\}s\{Q'\}} \quad (9.1)$$

$$\frac{b \wedge P \Rightarrow A(b)}{\vdash_a \{P\} \text{assume } b\{P \wedge b\}} \quad (9.2)$$

$$\frac{P \Rightarrow A(b) \quad P \Rightarrow b}{\vdash_a \{P\} \text{assert } b\{P \wedge b\}} \quad (9.3)$$

$$\frac{P \wedge b \Rightarrow A(\text{when } b \text{ do } v := e) \quad P \wedge b \Rightarrow Q[e/v]}{\vdash_a \{P\} \text{when } b \text{ do } v := e \{Q\}} \quad (9.4)$$

$$\frac{\vdash_a \{P\}s_1\{Q\} \quad \vdash_a \{Q\}s_2\{R\} \quad [Q \Rightarrow E(s_1) \vee Q \Rightarrow E(s_2)]}{\vdash_a \{P\}s_1; s_2\{R\}} \quad (9.5)$$

$$\frac{\vdash_a \{P\}s\{P\} \quad P \Rightarrow E(s)}{\vdash_a \{P\}s^*\{P\}} \quad (9.6)$$

$$\frac{\vdash_a \{P\}s_1\{Q\} \quad \vdash_a \{P\}s_2\{Q\}}{\vdash_a \{P\}s_1 \square s_2\{Q\}} \quad (9.7)$$

$$\frac{\vdash_a \{P\}s_1; s_2\{Q\} \quad \vdash_a \{P\}s_2; s_1\{Q\}}{\vdash_a \{P\}s_1 \parallel s_2\{Q\}} \quad (9.8)$$

$$\frac{}{\vdash_a \{P\} \text{skip}\{P\}} \quad (9.9)$$

9.2 Operational Semantics

$$\frac{\langle s_1, \sigma \rangle \rightarrow_{tid} \langle s'_1, \sigma' \rangle}{\langle s_1; s_2, \sigma \rangle \rightarrow_{tid} \langle s'_1; s_2, \sigma' \rangle} \quad (9.10)$$

$$\overline{\langle \text{skip}; s_2, \sigma \rangle \rightarrow_{tid} \langle s_2, \sigma \rangle} \quad (9.11)$$

$$\frac{\langle s_1, \sigma \rangle \rightarrow_{tid.0} \langle s'_1, \sigma' \rangle}{\langle s_1 \parallel s_2, \sigma \rangle \rightarrow_{tid} \langle s'_1 \parallel s_2, \sigma' \rangle} \quad (9.12)$$

$$\frac{\langle s_2, \sigma \rangle \rightarrow_{tid.1} \langle s'_2, \sigma' \rangle}{\langle s_1 \parallel s_2, \sigma \rangle \rightarrow_{tid} \langle s_1 \parallel s'_2, \sigma' \rangle} \quad (9.13)$$

$$\overline{\langle \text{skip} \parallel \text{skip}, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, \sigma \rangle} \quad (9.14)$$

$$\frac{b(\sigma, tid)}{\langle \text{assume } b, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, \sigma \rangle} \quad (9.15)$$

$$\frac{b(\sigma, tid)}{\langle \text{assert } b, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, \sigma \rangle} \quad (9.16)$$

$$\frac{\neg b(\sigma, tid)}{\langle \text{assert } b, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.17)$$

$$\frac{b(\sigma) \quad e(\sigma, tid) = x}{\langle \text{when } b \text{ do } v := e, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, [x/v]\sigma \rangle} \quad (9.18)$$

$$\overline{\langle s^*, \sigma \rangle \rightarrow_{tid} \langle s; s^*, \sigma \rangle} \quad (9.19)$$

$$\overline{\langle s^*, \sigma \rangle \rightarrow_{tid} \langle \text{skip}, \sigma \rangle} \quad (9.20)$$

$$\overline{\langle s_1 \square s_2, \sigma \rangle \rightarrow_{tid} \langle s_1, \sigma \rangle} \quad (9.21)$$

$$\overline{\langle s_1 \square s_2, \sigma \rangle \rightarrow_{tid} \langle s_2, \sigma \rangle} \quad (9.22)$$

$$\overline{\langle \text{wrong}, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.23)$$

$$\overline{\langle \text{wrong}^*, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.24)$$

$$\overline{\langle \text{wrong}; s, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.25)$$

$$\overline{\langle \text{wrong} \parallel s, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.26)$$

$$\overline{\langle s \parallel \text{wrong}, \sigma \rangle \rightarrow_{tid} \langle \text{wrong}, \sigma \rangle} \quad (9.27)$$

9.3 Proof of Soundness

Theorem 9.1. (Soundness Theorem)

*If $\vdash_a \{P\}s\{Q\}$ holds and $\sigma \models_{tid} P$
then $\langle s, \sigma \rangle \not\rightarrow_{tid}^*$ wrong*

Corollary 9.2. (Rightness Corollary)

If $\not\vdash_a \{P\}s\{Q\}$ then $s \neq \text{wrong}$.

Proof. This is a proof by inspection. There does not exist an Axiomatic Rule to handle the command wrong. Hence $\not\vdash_a \{P\}s\{Q\}$ □

Lemma 9.3. (Progress Lemma)

*If $\langle s, \sigma \rangle \rightarrow_{tid} \langle s', \sigma' \rangle$ and $\vdash_a \{P\}s\{Q\}$ and $\sigma \models_{tid} P$
then $\exists P'. \{\vdash_a \{P'\}s'\{Q\} \wedge \sigma' \models_{tid} P'\}$*

Proof. By induction on possible cases of $\langle s, \sigma \rangle \rightarrow_{tid} \langle s', \sigma' \rangle$.

Case 9.10:

$s = s_1; s_2 \quad s' = s'_1; s_2$

by $\vdash_a \{P\}s_1; s_2\{Q\}$ implies that $\vdash_a \{P\}s_1\{Q\}$ [9.5]

$\sigma \models_{tid} P$

By induction it follows that $\exists P'. \vdash_a \{P'\}s'_1\{Q\}$ & $\sigma' \models_{tid} P'$

Furthermore by 9.5 $\vdash_a \{P\}s_1; s_2\{R\} \Rightarrow \vdash_a \{Q\}s_2\{R\}$

so $\vdash_a \{P\}s'_1; s_2\{R\}$ & $\sigma' \models_{tid} P'$

Case 9.11:

$s = \text{skip}; s_2 \quad s' = s_2 \quad \sigma' = \sigma$

9.5 implies $\vdash_a \{P\}\text{skip}\{Q\}$ & $\vdash_a \{Q\}s_2\{R\}$ and 9.9 implies that $Q = P$

so take $P' = P$. $\vdash_a \{P'\}s_2\{R\}$ and it was given that $\sigma' \models_{tid} P'$

Case 9.12:

To be proven.

Case 9.13:

To be proven.

Case 9.14:

To be proven.

Case 9.15:

$s = \text{assume } b \quad s' = \text{skip} \quad b(\sigma, tid) \quad \sigma' = \sigma$

By $\vdash_a \{P\}\text{assume } b\{Q\} \quad Q = P \wedge b$. [9.2]

As $\sigma \models_{tid} P$ it follows that $\sigma' \models_{tid} P$ and by 9.2 $b(\sigma, tid)$ so it follows that $b(\sigma', tid)$.

Hence $\sigma' \models_{tid} P \wedge b$.

Take $P' = P \wedge b = Q$. Clearly $\sigma' \models_{tid} P'$ & $\vdash_a \{P'\}\text{skip}\{P'\}$

Case 9.16:

Similar to case 2.6. $s = \text{assert } b \quad s' = \text{skip} \quad b(\sigma, tid) \quad \sigma' = \sigma$

By 9.9 $Q = P \wedge b$.

As $\sigma \models_{tid} P$ it follows that $\sigma' \models_{tid} P$ and by [9.3 $b(\sigma, tid)$ it follows that $b(\sigma', tid)$.

Hence $\sigma' \models_{tid} P \wedge b$. Take $P' = P \wedge b = Q$. clearly $\sigma' \models_{tid} P'$ & $\vdash_a \{P'\}\text{skip}\{P'\}$

Case 9.17:

$s = \text{assert } b \quad s' = \text{skip} \quad \neg b(\sigma, tid) \quad \sigma' = \sigma \quad \sigma \models_{tid} P$

Hence $\sigma' \models_{tid} P$ and $\neg b(\sigma', tid)$ However by 9.2 as $\vdash_a \{P\} \text{assert } b \{P \wedge b\}$ it follows that $P \Rightarrow b$. This is a contradiction.

Case 9.18:

$s = \text{when } b \text{ do } v := e \quad s' = \text{skip} \quad e(\sigma, tid) = x \quad \sigma' = [x/v]\sigma$

Take $P' = Q$. Clearly $\vdash_a \{P'\} \text{skip}\{P'\}$

Lemma (to be proven)

If $\sigma' = \sigma[v := e]$ and $\sigma \models_{tid} Q[e/v]$

then $\sigma' \models Q$

Case 9.19:

$s = s^* \quad s' = s; s^* \quad \sigma' = \sigma$

By 9.6 $P = Q \ \& \ \vdash_a \{P\} s \{P\} \ \& \ P \Rightarrow E(s)$.

Together with $\vdash_a \{P\} s^* \{P\}$, 9.5 implies that $\vdash_a \{P\} s; s^* \{P\}$. It is also clear that $\sigma' \models P$

Case 9.20:

$s = s^* \quad s' = \text{skip} \quad \sigma' = \sigma$

Choose $P' = P$. $\vdash_a \{P\} \text{skip}\{P\}$ and clearly $\sigma \models P'$.

Case 9.21:

$s = s_1 \square s_2 \quad s' = s_1 \quad \sigma' = \sigma$

By 9.7 $\vdash_a \{P\} s_1 \square s_2 \{Q\} \Rightarrow \vdash_a \{P\} s_1 \{Q\}$ and clearly $\sigma \models P$.

Case 9.22:

Similar to case 2.12.

Case 9.23:

$s = \text{wrong}$

$\vdash_a \{P\} \text{wrong}\{Q\}$. By the rightness corollary that assertion is false. Contradiction. Hence this rule would never be applicable.

Case 9.25:

$s = \text{wrong}^*$

$\vdash_a \{P\} \text{wrong}^* \{Q\}$ and wish to prove that $\vdash_a \{P\} \text{wrong}\{Q\}$ is valid. By the rightness corollary that assertion is always false. Contradiction. Hence this rule would never be applicable.

Case 9.10:

$s = \text{wrong}; s$

$\vdash_a \{P\} \text{wrong}; s \{Q\}$ and wish to prove that $\vdash_a \{P\} \text{wrong}\{Q\}$ valid. By the rightness corollary that assertion is always false. Contradiction. Hence this rule would never be applicable.

Case 9.12:

$s = \text{wrong} \parallel s_2$

$\vdash_a \{P\} \text{wrong} \parallel s_2 \{Q\} \Rightarrow \vdash_a \{P\} \text{wrong}; s_2 \{Q\}$. This then reduces to case 2.16. Hence this rule would never be applicable.

Case 9.27:

Similar to Case 2.17 □

Proof of Soundness. Assume the Progress Lemma. Suppose that $\exists s'$ such that $\langle s, \sigma \rangle \rightarrow_{tid}^* \langle s', \sigma' \rangle$. $\vdash_a \{P\} s \{Q\}$ holds and $\sigma \models_{tid} P$

No matter how many times s is reduced the resulting program s' is valid. Or $\vdash_a \{P\} s' \{Q\}$. This is true by induction.

Base case: the 0th reduction. $s = s'$ and $\sigma = \sigma'$. It is vacuously true that $\vdash_a \{P\} s' \{Q\}$ holds and $\sigma' \models_{tid} P$ as it is true for s and σ .

Inductive case: the n^{th} reduction. We may assume by induction that the $n-1$ program reductions preserves both the validity and produces a state which applies the precondition of the new program. In other words $\forall s' \text{ such that } \langle s, \sigma \rangle \xrightarrow{tid}^{n-1} \langle s'', \sigma'' \rangle$. we know that $\vdash_a \{P''\}s''\{Q''\}$ holds and $\sigma'' \models_{tid} P''$. We also know that there exists a s'' such that $\langle s'', \sigma'' \rangle \xrightarrow{tid} \langle s', \sigma' \rangle$ because s' is the result of n reductions of the original program s . By the Progress Lemma we know that $\exists P'$ such that $\vdash_a \{P\}s'\{Q\}$ holds and $\sigma' \models_{tid} P$. Done.

By the rightness corollary s' is also not the program wrong. □